input_dataset — List of vectors or lists with integers representing words in review. Every individual vector / List represents a review.

target_dataset — list of 0's and 1's
negative review. Positive review

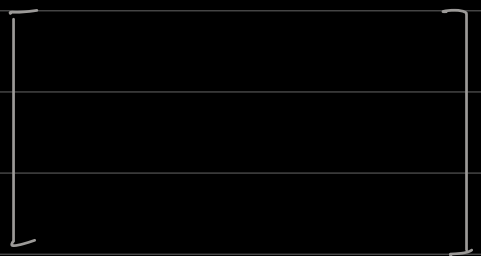
images ↓ input pixels in one image. ↓

Matrix (1000, 784) — gives one vector/ image

Weights_0_1 (784, 40) — 40 (hidden layer and # of neurons = 40
↑ rows ↑ columns

Weights_1_2 (40, 10) — 10 (outputs)
↑ rows ↑ columns

processing all images / vectors — 350 times to reduce error.

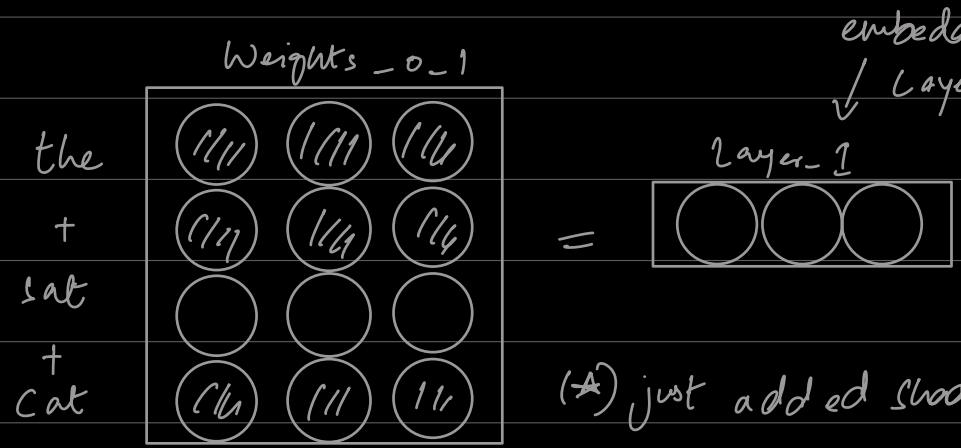


matrix of reviews and every vector / row represents a review and every review consist of

*) weights_0_1 and weight_1_2 (Linear Layers) bunch of 0's and 1's mode using one hot encoding vectors of vocab or words in a review

*) We can take a bit of shortcut to Layer_1 by replacing the first linear layer weights_0_1 with an embedding Layer.

***) Taking a vector of 1's and 0's is equivalent to summing several rows of a matrix. ∴ Instead of vector multiplication — sum relevant rows of weights_0_1



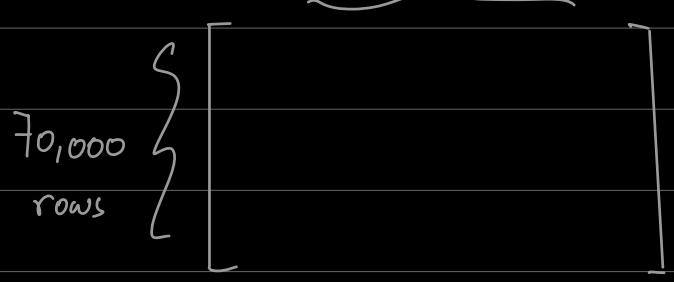
(*) just added shaded rows rather matrix vector multiplication

24000 - reviews ⇒ 24000 dimensions

Total vocab / words - 70,000

weights_0_1 ⟶ (70,000, 100)

weights_0_1 ⟶ (100, 1)



input_dataset ⟶ (24,000, 70,000)

reviews

cols

24,000, 70,00

processing (1, 70,000) - vector / review

(70,000, 100) - weight_matrix

(1, 70,000) · (70,000, 100) = (1, 100)

vector of inputs / neurons

Matrix

vector

inputs

weights

70,000 · 70,000 } takes place 100 times since

100 cols → and every multiplication yields a neuron.

*) However, in this case we would not multiply and simply add rows of weight matrix corresponding one input vector of 1 × 70,000 → giving us 1 × 100 vector or embedding layer.

*) [1 × 100] vector then gets multiplied to 100×1 matrix or vector. — which yields an output of 1 or 0 — where 1 represents positive.
and 0 represents negative

⇒ Training neural network to predict if review is positive or negative.

```
import sys
f = open('reviews.txt')
raw_reviews = f.readlines()
f.close()
```

```
f = open('labels.txt')
raw_labels = f.readlines()
f.close()

tokens = list(map(lambda x: set(x.split(" ")),
                  raw_reviews)) # list of sets, sets representing individual reviews.

vocab = set
for sent in tokens: # iterating through individual set/review.
    for word in sent:
        if len(word) > 0:
            vocab.add(word)
vocab = list(word)  # all words

word2index = {} # word to integer/index
for i, word in enumerate(vocab):
    word2index[word] = i

input_dataset = list()
for sent in tokens:
    sent_indices = list()
    for word in sent:
```

pre-processing

```
            try:
                sent_indices.append(word2index[word])
            except:
                " "
        input_dataset.append(list(set(sent_indices)))

target_dataset = list()
for label in raw_labels:
    if label == 'positive\n':
        target_dataset.append(1)
    else:
        target_dataset.append(0)
```

input_dataset is list of vectors / lists with integers/ indices representing words in a review.
↳ a vector or list of words as integers/ indices in a review.

```
import numpy as np
np.random.seed(1)

def sigmoid(x):
    return 1/(1 + np.exp(-x))

alpha, iterations = (0.01, 2)   # iterations = epochs
hidden_size = 100               # features

# matrix of nums b/w -0.1 to 0.1
weights_0_1 = 0.2 * np.random.random((len(vocab), hidden_size)) - 0.1
weights_1_2 = 0.2 * np.random.random((hidden_size, 1)) - 0.1

correct, total = (0, 0)
for epoch in range(iterations):
    # Trains on first 24,000 reviews.
    for i in range(len(input_dataset) - 1000):
        # Loading a review and a label.
        x, y = (input_dataset[i], target_dataset[i])
        layer_1 = sigmoid(np.sum(weights_0_1[x], axis=0))
        layer_2 = sigmoid(np.dot(layer_1, weights_1_2))

        # vector of 100 cols or features.
        # truth or label (0 or 1)
        layer_2_delta = layer_2 - y
        # Backprop gator
        layer_1_delta = layer_2_delta.dot(weights_1_2.T)
```

Linear + Softmax a output blw 0 to 1 signifying Probability

```
weights_0_1[x] -= layer_1_delta * alpha
weights_1_2 -= np.outer(layer_1, layer_2_delta) * alpha
                  (100 neurons in hidden layer)

if (np.abs(layer_2_delta) < 0.5):
        correct += 1
total += 1
if (i % 10 == 9):
    progress = str(i / float(len(input_dataset)))

    sys.stdout.write('progress:' + progress[2:4] \
        + '.' + progress[4:6] \
        + '% training Accuracy:' \
        + str(correct / float(total)) \
        + '%')

print()
```

testing accuracy:

```
correct, total = (0, 0)
for i in range(len(input_dataset) - 1000, len(input_dataset)):
```

```
x = input_dataset[i]  # review
y = target_dataset[i] # label / actual output

layer_1 = sigmoid(np.sum(weights_0_1[x], axis=0))
layer_2 = sigmoid(np.dot(layer_1, weights_1_2))

# if(np.abs(layer_2 - y) < 0.5):
if output          correct += 1
Prob > 0.5         total += 1
that means
guessed
correctly.         print("Test accuracy:" + str(correct/float(total)))
```

*) Testing accuracy ⟶ 0.849 — accurate 84.9% accuracy.

*) Network identifies words that have either a positive negative correlation.

*) Fundamentally, hidden layers are about grouping data points from a previous layer into n groups (where n — number of neurons in the hidden layer).

Each hidden neuron takes in a datapoint and answers the question "is this datapoint in my group"

*) An input data point grouping is useful if it does two things. First, the grouping must be useful to the prediction of an output layer.

*) Second, a grouping is useful if it's an actual phenomenon in the data that you care about. Good groupings pick up on phenomenon that are useful linguistically.

*) If you can construct two examples with an identical hidden layer, one with the pattern you find interesting and one without, the network is unlikely to find that pattern — thinking of groupings using not or negation. (example).

*) A hidden layer fundamentally groups the previous layer's data.

*) At a granular level, each neuron classifies a datapoint as either subscribing or not subscribing to its group.

(movie reviews)

*) At a higher level, two datapoints are similar if they subscribe to many of the same groups.

*) Finally, two inputs (words) are similar if weights linking them to various hidden neurons (a measure of each word's group affinity) are similar.

*) Words that have similar predictive power should subscribe to similar groups (hidden neuron configurations).

*) Words that correlate with similar labels (positive or negative) will have similar weights connecting them to various neurons.

*) This phenomenon can be noticed by taking a particular pos or neg word and searching for the

Other word with the most similar weight values.

*) Words that subscribe to similar groups will have similar predictive power for positive or negative labels. ∴ Words that subscribe to similar groups, having similar weight values, will also have similar meaning.

*) To figure out which words are most similar to a target term, you compare each word's vector (row of the matrix) to that of the target term. Using dot product works as well for similarity.

```
from collections import Counter
import math

def similar(target='beautiful'):
    target_index = word2index[target]
    scores = Counter()
    for word,index in word2index.items():
        raw_difference = weights_0_1[index] - (weights_0_1[target_index])
        squared_difference = raw_difference * raw_difference
        scores[word] = -math.sqrt(sum(squared_difference))

    return scores.most_common(10)
```

Euclidian distance.

This allows you to easily query for the most similar word (neuron) according to the network:

```
print(similar('beautiful'))
```

```
[('beautiful', -0.0),
 ('atmosphere', -0.70542101298),
 ('heart', -0.7339429768542354),
 ('tight', -0.7470388145765346),
 ('fascinating', -0.7549291974),
 ('expecting', -0.759886970744),
 ('beautifully', -0.7603669338),
 ('awesome', -0.76647368382398),
 ('masterpiece', -0.7708280057),
 ('outstanding', -0.7740642167)]
```

```
print(similar('terrible'))
```

```
[('terrible', -0.0),
 ('dull', -0.760788602671491),
 ('lacks', -0.76706470275372),
 ('boring', -0.7682894961694),
 ('disappointing', -0.768657),
 ('annoying', -0.78786389931),
 ('poor', -0.825784172378292),
 ('horrible', -0.83154121717),
 ('laughable', -0.8340279599),
 ('badly', -0.84165373783678)]
```

Since, the network has only two labels, the input terms are grouped according to which label they tend to predict.

•) input_size: # of expected features in the input x (depth)
input of 20 values in sequence and 1 in input size features (depth)

•) hidden_size: # of features / columns the output of RNN and hidden state will have.

•) batch_first: if True — input and output tensors will (batch_size, seq, feature) — Tensor.
↑ batch_size as 1st dimension.

Seq_length = 20 # since RNNs work with sequence data.

— Create sample input and target sequence of data points of lenght 20.

— Sequence length — # of words in a sentence.

21 values for x-axis all the way up to pi.

```
time_steps = np.linspace(0, np.pi, seq_length+1)
data = np.sin(time_steps) # y values.
data.resize((seq_length+1), 1) # size becomes
(seq_length+1, 1), adds an input_size dimension

x = data[:-1] # input data   } 20 datapoints
y = data[1:] # target data.  } y is basically x
                             but shifted
                             one timestep
                             in the future.
plt.plot(time_steps[1:], x, 'r.')   # x = input
plt.plot(time_steps[1:], y, 'b.')   # y = target
```

Recurrent layer — documentation — responsible for calculating a hidden states based on its input.

```
Class RNN(nn.Module):
    def __init__(self, in_size, ou_size, hid_dim, n_lay)
        super(RNN, self).__init__()

        self.hidden_dim = hid_dim   # vector components defines number of features the output of this RNN will have.
        Self.rnn = nn.RNN(in_size, hid_dim, n_lay, batch_first=True)

        Self.fc = nn.Linear(hid_dim, ou_size)
                            # output of   # of outputs.
```

RNN

hidden state.

```
def forward(self, x, hidden):

    batch_size = x.size(0)
    r_out, hidden = self.rnn(x, hidden)
# shape r_out to (batch_size * seq_length, hidden_dim)
    r_out = r_out.view(-1, self.hidden_dimension)
    output = self.fc(r_out)
    return output, hidden
```

cols

input — [batch, seq_l, feature] [1, 20, 1]

output — [20, 1]

Batch size, output

(20, 10)

Hidden — [2, 10, 1]

2 ↑ layers, 10 ↑ dimension

(1, 1) ∘ (1 ∘ 32) → (1, 32) ← hidden or state output neurons

weight matrix (20, 1)?

Vector (1, 32) / 20, 32

r_out = (20, 32) ⇒ 20 r's, 32 c's

1 * 20

W → (32, 1)

I — (20, 32)

W → (32, 1)

(20, 1) where every column is Predicted value for a sequence.

*) For every iteration training on a sequence, but memory None first but after 1st iteration added..

input sequence ↓ 20 → □ → 20 ↓ output sequence.

- Deep Learning framework provides support for automatic back propogation and automatic optimization. These features let you specify only the forward propogation code of a model with the framework taking care of backpropogation and weight updates automatically.
- Providing high level interfaces to common layers and loss functions.