CHAPTER 2: MATHEMATICAL BUILDING BLOCK OF NN

**A FIRST LOOK AT NN**

1. **NIST (MNIST or National Institute of Standards and Technology):** like hello world of DL => verify if algorithm works as intended.

**Loading MNIST dataset**

**from keras.datasets import mnist**

**(training\_set), (test\_set) = mnist.load\_data()**

1. In classification: category = class, datapoints = samples, class w/ specific sample = label

Ex: Images will be encoded in Numpy arrays, label will be a array of digits (0 to 9) => handwriting problem

1. Workflow:
   1. Feed NN training data
   2. NN will learn to associate images and labels
   3. Use it on test data
2. The core building block of NN is the layer, a data-processing module like a filter for data
3. **Layers** extract representations out of the data fed into them

Combine simple layers => form of progressive **data distillation**

1. A DL model is like a sieve for data processing, made of succession of increasingly refinded data filter- the layers

**network architecture**

**from keras import models**

**from keras import layers**

**network = models.Sequential()**

**network.add(layers.Dense(512, activation='relu', input\_shape=(28 \* 28,)))**

**network.add(layers.Dense(10, activation='softmax'))**

1. **2 Dense** layers (densely/fully connected), the 2nd layers is 10-way softmax layer => return an **array** of 10 probability scores (summing to 1). Each score => probability that current digit image belongs to.
2. To make NN ready for training: 3 more things: **loss function/optimizer/metric to monitor process**

**Compilation step**

**network.compile(optimizer='rmsprop',**

**loss='categorical\_crossentropy',**

**metrics=['accuracy'])**

1. Before training, preprocess the data by reshape it into the shape of network expects, and scaling it so the value are in [0,1] interval. [uint8] with [0,255]. Transform into float32 array of shape (60000, 28\*28) w/ value bw [0,1]

**Prepare image Data**

**train\_images = train\_images.reshape((60000, 28 \* 28))**

**train\_images = train\_images.astype('float32') / 255**

**test\_images = test\_images.reshape((10000, 28 \* 28))**

**test\_images = test\_images.astype('float32') / 255**

**Prepare the labels**

**from keras.utils import to\_categorical**

**train\_labels = to\_categorical(train\_labels)**

**test\_labels = to\_categorical(test\_labels)**

**network.fit(train\_images, train\_labels, epochs=5, batch\_size=128)**

**>>> test\_loss, test\_acc = network.evaluate(test\_images, test\_labels)**

**>>> print('test\_acc:', test\_acc)**

10. Gap between training data and test data is **Overfitting**

11. **Tensors:** data stored in multidimensional Numpy arrays. Basic data structure for ML. Tensor is a container for data (almost always numerical data). Matrixes are 2D tensors.

12. **Scalars** (0D tensors): a tensor that contains 1 number is scalar (or scalar tensor). In **Numpy**, a float32 or float64 is scalar tensor (or scalar array).

Can call attribute of numpy array by **ndim**

Number of axes of a tensor is called **rank**

**>>> import numpy as np**

**>>> x = np.array(12)**

**>>> x**

**array(12)**

**>>> x.ndim**

**0**

13. **Vectors** (1D tensors):

**>>> x = np.array([12, 3, 6, 14])**

**>>> x**

**array([12, 3, 6, 14])**

**>>> x.ndim**

**1**

This vector has five entries and called 5D vector. 5D vector is not 5D tensor.

5D vector has only 1 axis and 5 dimensions along its axis, where 5D tensor will has 5 axes.

14. **Matrices (2D tensors)**

**>>> x = np.array([[5, 78, 2, 34, 0],**

**[6, 79, 3, 35, 1],**

**[7, 80, 4, 36, 2]])**

**>>> x.ndim**

**2**

Is an array of vectors.

15. **3D tensors and higher**

By packing 3D tensors in an array, you will have 4D tensor, and so on. In DL, will work with 0D to 4D, maybe 5D in video data.

16. **Key attributes:** a tensor is defined by three key attributes:

- **Number of axes (rank):** tensor’s ndim in Python libraries such as Numpy

-  **Shape:** - a tuple of integers that decribes how many dimension the tensor has along each axis

Ex: matrix shape (3,5), 3D tensor (3,3,5), vector (5, ), and scalar has empty shape ().

-  **Data type: dtype of** libraries (can be float32, unit8, float64, maybe char tensor)

17. **Tensor slicing:** select specific elements in a tensor

**>>> my\_slice = train\_images[10:100]**

**>>> print(my\_slice.shape)**

**(90, 28, 28)**

**>>> my\_slice = train\_images[10:100, :, :]**

**>>> my\_slice.shape**

**(90, 28, 28)**

**>>> my\_slice = train\_images[10:100, 0:28, 0:28]**

**>>> my\_slice.shape**

**(90, 28, 28)**

**>>> my\_slice = train\_images[:, 14:, 14:] #bottom-right corner of all images**

**>>> my\_slice = train\_images[:, 7:-7, 7:-7] #14x14 pixels centered in the middle**

18. In general, axis 0 (first) will be **sample axis (**or **sample dimension).** In MNIST example, samples are images of digits. **DL model break the data into small batches (don’t process all at once).**

Also, when taking about batch, the first axis (0) will be called **batch axis,** or **batch dimension**

**batch = train\_images[:128]**

**And here’s the next batch:**

**batch = train\_images[128:256]**

**And the n th batch:**

**batch = train\_images[128 \* n:128 \* (n + 1)]**

19. **Real world example**

**- Vector data— 2D tensors of shape (samples, features)**

**- Timeseries data or sequence data— 3D tensors of shape (samples, timesteps,**

**features)**

**- Images— 4D tensors of shape (samples, height, width, channels) or (samples,**

**channels, height, width)**

**- Video— 5D tensors of shape (samples, frames, height, width, channels) or**

**(samples, frames, channels, height, width)**

20. **Vector data:** each single data point can be encoded as a vector => batch of data is 2D tensor. And, first axis is **sample axis** and second axis is **feature axis.**

21. **Timeseries data or sequence data:** when time matters in your data => store it in explicit time axis. The time axis is always the second axis (index 1).

22. **Image data:** always 3D tensor, w/ one dimensional color channel for grayscale images

A batch of 128 grayscale images of

size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1) , and a

batch of 128 color images could be stored in a tensor of shape (128, 256, 256, 3)

Tensorflow use (samples, height, width, color\_depth) or channels-last convention

23. **Video data:** one of the few need 5D tensors. A video is sequence of frames, each frame is a color image. => batch of frame of color image => 5D

(samples, frames, height, width, color\_depth)

For instance, a 60-second, 144 × 256 YouTube video clip sampled at 4 frames per

second would have 240 frames. A batch of four such video clips would be stored in a

tensor of shape (4, 240, 144, 256, 3) . That’s a total of 106,168,320 values! If the

dtype of the tensor was float32 , then each value would be stored in 32 bits, so the

tensor would represent 405 MB. Heavy! Videos you encounter in real life are much

lighter, because they aren’t stored in float32 , and they’re typically compressed by a

large factor (such as in the MPEG format).

24. **Tensor operations (gear of NN):** (add, multiply…)

keras.layers.Dense(512, activation='relu') #keras layer instance

#take input 2D tensor and return 2D tensor

output = relu(dot(W, input) + b) #W is 2D tensor, b is vector

#dot product bw 2 2D tensor, addition, and relu(x)

25. **Element-wise operations:** likerelu or add(+) because it is applied independently to each entry in the tensors => highly amenable to massively parallel implementations (**vectorized** implementation)

def naive\_relu(x):

assert len(x.shape) == 2

x is a 2D Numpy tensor.

x = x.copy()

for i in range(x.shape[0]):

for j in range(x.shape[1]):

x[i, j] = max(x[i, j], 0)

return x

def naive\_add(x, y):

assert len(x.shape) == 2

assert x.shape == y.shape

x = x.copy()

for i in range(x.shape[0]):

for j in range(x.shape[1]):

x[i, j] += y[i, j]

return x

26. **Boradcasting** when add 2 different size tensor, the smaller one will be broadcasted to match the larger tensor. Steps: - axes (or broadcast axes) are added to match the ndim of larger tensor.

- The smaller tensor is repeated alongside the new axes to match full size

Let’s look at a concrete example. Consider X with shape (32, 10) and y with shape

(10,) . First, we add an empty first axis to y , whose shape becomes (1, 10) . Then, we

repeat y 32 times alongside this new axis, so that we end up with a tensor Y with shape

(32, 10) , where Y[i, :] == y for i in range(0, 32) . At this point, we can proceed to

add X and Y , because they have the same shape.

def naive\_add\_matrix\_and\_vector(x, y):

assert len(x.shape) == 2

assert len(y.shape) == 1

assert x.shape[1] == y.shape[0]

x = x.copy()

for i in range(x.shape[0]):

for j in range(x.shape[1]):

x[i, j] += y[j]

return x

With broadcasting, you can generally apply two-tensor element-wise operations if one

tensor has shape (a, b, ... n, n + 1, ... m) and the other has shape (n, n + 1, ... m) . The

broadcasting will then automatically happen for axes a through n - 1 .

import numpy as np

x = np.random.random((64, 3, 32, 10))

y = np.random.random((32, 10))

z = np.maximum(x, y)

26. **Tensor dot** (or **tensor product**) is most common and useful operation. Have entry input tensors. In Numpy or Keras, “z = np.dot(x,y)”

def naive\_vector\_dot(x, y): # x and y are Numpy vectors.

assert len(x.shape) == 1

assert len(y.shape) == 1

assert x.shape[0] == y.shape[0]

z = 0.

for i in range(x.shape[0]):

z += x[i] \* y[i]

return z

def naive\_matrix\_vector\_dot(x, y): # x is matrix, y is vector

assert len(x.shape) == 2

assert len(y.shape) == 1

assert x.shape[1] == y.shape[0]

z = np.zeros(x.shape[0])

for i in range(x.shape[0]):

for j in range(x.shape[1]):

z[i] += x[i, j] \* y[j]

return z

def naive\_matrix\_vector\_dot(x, y): # OR

z = np.zeros(x.shape[0])

for i in range(x.shape[0]):

z[i] = naive\_vector\_dot(x[i, :], y)

return z

def naive\_matrix\_dot(x, y):

assert len(x.shape) == 2

assert len(y.shape) == 2

assert x.shape[1] == y.shape[0]

z = np.zeros((x.shape[0], y.shape[1]))

for i in range(x.shape[0]):

for j in range(y.shape[1]):

row\_x = x[i, :]

column\_y = y[:, j]

z[i, j] = naive\_vector\_dot(row\_x, column\_y)

return z

Dot product between higher-dimensional, follow the same rule

(a, b, c, d) . (d, ) → (a, b, c)

(a, b, c, d) . (d, e) → (a, b, c, e)

27. **Tensor reshaping**

train\_images = train\_images.reshape((60000, 28 \* 28))

It means rearranging its rows and columns to match the target shape.

>>> x = np.array([ [0., 1.],

[2., 3.],

[4., 5.]])

>>> print(x.shape)

(3, 2)

>>> x = x.reshape((6, 1))

>>> x

array([ [ 0.],

[ 1.],

[ 2.],

[ 3.],

[ 4.],

[ 5.]])

>>> x = x.reshape((2, 3))

>>> x

array([ [ 0., 1., 2.],

[ 3., 4., 5.]])

28. **Geometric Interpretation of tensor operations**

In general, elementary geometric operations such as affine transformations, rotations,

scaling, and so on can be expressed as tensor operations. For instance, a rotation of a

2D vector by an angle theta can be achieved via a dot product with a 2 × 2 matrix

R = [u, v] , where u and v are both vectors of the plane: u = [cos(theta),

sin(theta)] and v = [-sin(theta), cos(theta)] .

29. **Geometric interpretation of DL** Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data manifolds.

30. **Gradient-based optimization**

output = relu(dot(W, input) + b)

W and b are tensors that are attributes of layer (**weights** or **trainable parameter** of the layer/ OR the kernel and bias attributes). Getting this information from training data.

31. **Random initialization** random value for W and b at first.

This happens within what’s called a training loop, which works as follows. Repeat

these steps in a loop, as long as necessary:

1/ Draw a batch of training samples x and corresponding targets y .

2/ Run the network on x (a step called the forward pass) to obtain predictions y\_pred .

3/ Compute the loss of the network on the batch, a measure of the mismatch

between y\_pred and y .

4/ Update all weights of the network in a way that slightly reduces the loss on this

batch.

32. All operation used in the network are **differentiable**, and compute the **gradient** of the loss with regard to the network’s coefficients. You can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss.

33. **Derivative of a tensor operation: the gradient**

y\_pred = dot(W, x)

loss\_value = loss(y\_pred, y)

If the data inputs x and y are frozen, then this can be interpreted as a function map-

ping values of W to loss values:

loss\_value = f(W)

Let’s say the current value of W is W0 . Then the derivative of f in the point W0 is a tensor

gradient(f)(W0) with the same shape as W , where each coefficient gradient(f)

(W0)[i, j] indicates the direction and magnitude of the change in loss\_value you

observe when modifying W0[i, j] . **That tensor gradient(f)(W0) is the gradient of**

**the function f(W) = loss\_value in W0 .**

34. **Stochastic gradient descent:** Finding minimal value of a differentiable function. For a NN, means finding analytically the combination of weight values that yields the smallest possible loss function. Solving the equation **gradient (f) (W) = 0 for W**. This is polynomial equation of N vars, N is the number of coefficients in the network.

1/ Draw a batch of training samples x and corresponding targets y .

2/ Run the network on x to obtain predictions y\_pred .

3/ Compute the loss of the network on the batch, a measure of the mismatch

between y\_pred and y .

4/ Compute the gradient of the loss with regard to the network’s parameters (a

backward pass).

5/ Move the parameters a little in the opposite direction from the gradient—for

example W -= step \* gradient —thus reducing the loss on the batch a bit.

**mini-batch stochastic gradient descent (mini-batch SGD)**

**[NO IDEA]**

**Note that a variant of the mini-batch SGD algorithm would be to draw a single sam-**

**ple and target at each iteration, rather than drawing a batch of data. This would be**

**true SGD (as opposed to mini-batch SGD ). Alternatively, going to the opposite extreme,**

**you could run every step on all data available, which is called batch SGD . Each update**

**would then be more accurate, but far more expensive. The efficient compromise**

**between these two extremes is to use mini-batches of reasonable size.**

**Additionally, there exist multiple variants of SGD that differ by taking into account**

**previous weight updates when computing the next weight update, rather than just**

**looking at the current value of the gradients. There is, for instance, SGD with momen-**

**tum, as well as Adagrad, RMSP rop, and several others. Such variants are known as opti-**

**mization methods or optimizers. In particular, the concept of momentum, which is used in**

**many of these variants, deserves your attention. Momentum addresses two issues with**

**SGD : convergence speed and local minima. Consider figure 2.13, which shows the**

**curve of a loss as a function of a network parameter.**

**Momentum is implemented by moving**

**the ball at each step based not only on the current slope value (current acceleration)**

**but also on the current velocity (resulting from past acceleration). In practice, this**

**means updating the parameter w based not only on the current gradient value but also**

**on the previous parameter update, such as in this naive implementation:**

past\_velocity = 0.

#Constant momentum factor

momentum = 0.1

#Optimization loop

while loss > 0.01:

w, loss, gradient = get\_current\_parameters()

velocity = past\_velocity \* momentum + learning\_rate \* gradient

w = w + momentum \* velocity - learning\_rate \* gradient

past\_velocity = velocity

update\_parameter(w)

35. **Chaining derivatives: the Backpropagation algorithm**

Calculus tells us that such a chain of functions can be derived using the following iden-

tity, called the chain rule: f(g(x)) = f'(g(x)) \* g'(x) . Applying the chain rule to the

computation of the gradient values of a neural network gives rise to an algorithm called **Backpropagation** (also sometimes called **reverse-mode differentiation**). Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

**Chapter summary**

- Learning means finding a combination of model parameters that mini-

mizes a loss function for a given set of training data samples and their cor-

responding targets.

- Learning happens by drawing random batches of data samples and their

targets, and computing the gradient of the network parameters with

respect to the loss on the batch. The network parameters are then moved

a bit (the magnitude of the move is defined by the learning rate) in the

opposite direction from the gradient.

- The entire learning process is made possible by the fact that neural net-

works are chains of differentiable tensor operations, and thus it’s possible

to apply the chain rule of derivation to find the gradient function map-

ping the current parameters and current batch of data to a gradient value.

- Two key concepts you’ll see frequently in future chapters are loss and opti-

mizers. These are the two things you need to define before you begin feed-

ing data into a network.

- The loss is the quantity you’ll attempt to minimize during training, so it

should represent a measure of success for the task you’re trying to solve.

- The optimizer specifies the exact way in which the gradient of the loss will

be used to update parameters: for instance, it could be the RMSP rop opti-

mizer, SGD with momentum, and so on.