Engineering Risk Analysis Group
Department of Civil, Geo and Environmental Engineering
Technical University of Munich

Seminar on Elements of Machine Learning

Feed-Forward Neural Network

Ahmad M. Belbeisi

Report for the

Seminar on Elements of Machine Learning

at the Department of Civil, Geo and Environmental Engineering of the Technical University of Munich.

Examiner:
Prof. Dr. Daniel Straub

Supervisor:
Oindrila Kanjilal

Submitted:
Munich, 01.04.2021

Contents

1 Neural Networks

1.1 Feedforward Neural Network

1.1.1 Logistic Regression as Neural Network

Logistic regression from the point of view of neural network. For a binary classification problem, for example, spam classifier, given m samples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$, we need to use the input feature $x^{(i)}$ (They may be the frequency of words such as "money," special characters like dollar signs, the use of capital letters in the message, etc.) to predict the output $y^{(i)}$ (if it is a spam email). Assume that for each sample i, there are n_x input features. Then we have:

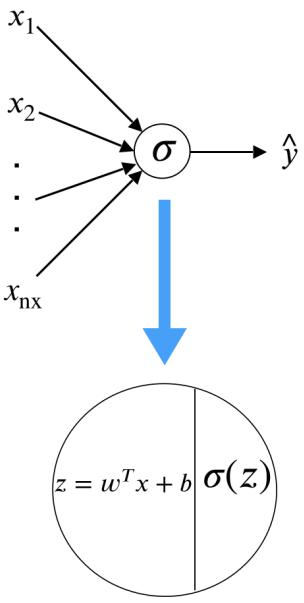
$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \vdots & \vdots & \vdots & \vdots \\ x_{n_x}^{(1)} & x_{n_x}^{(2)} & \cdots & x_{n_x}^{(m)} \end{bmatrix} \in \mathbb{R}^{n_x \times m} \tag{1.1}$$

$$y = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}] \in \mathbb{R}^{1 \times m}$$

To predict if sample i is a spam email, we first get the inactivated **neuro** $z^{(i)}$ by a linear transformation of the input $x^{(i)}$, which is $z^{(i)} = w^T x^{(i)} + b$. Then we apply a function to "activate" the neuro $z^{(i)}$ and we call it "activation function". In logistic regression, the activation function is sigmoid function and the "activated" $z^{(i)}$ is the prediction:

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$. The following figure summarizes the process:



There are two types of layers; the last layer connects directly to the output. All the rest are middle layers. Depending on your explanation, we call it a "0-layer neural network" where the layer count only considers intermediate layers. To train the model, you need a cost function defined as equation 1.2.

$$J(w,b) = \frac{1}{m}\Sigma_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) \tag{1.2}$$

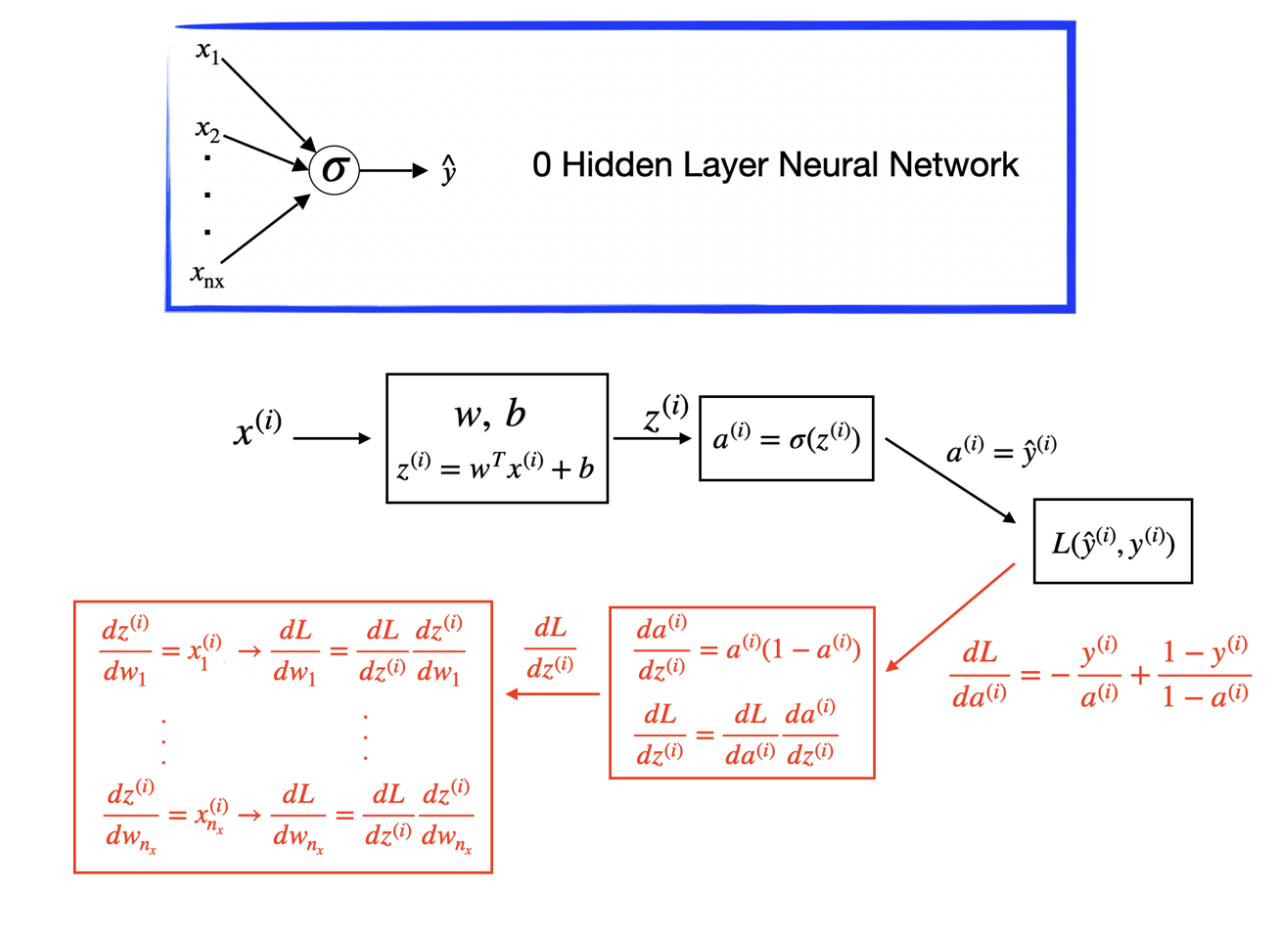
where

$$L(\hat{y}^{(i)}, y^{(i)}) = -y^{(i)} log(\hat{y}^{(i)}) - (1-y^{(i)}) log(1-\hat{y}^{(i)})$$

To fit the model is to minimize the cost function.

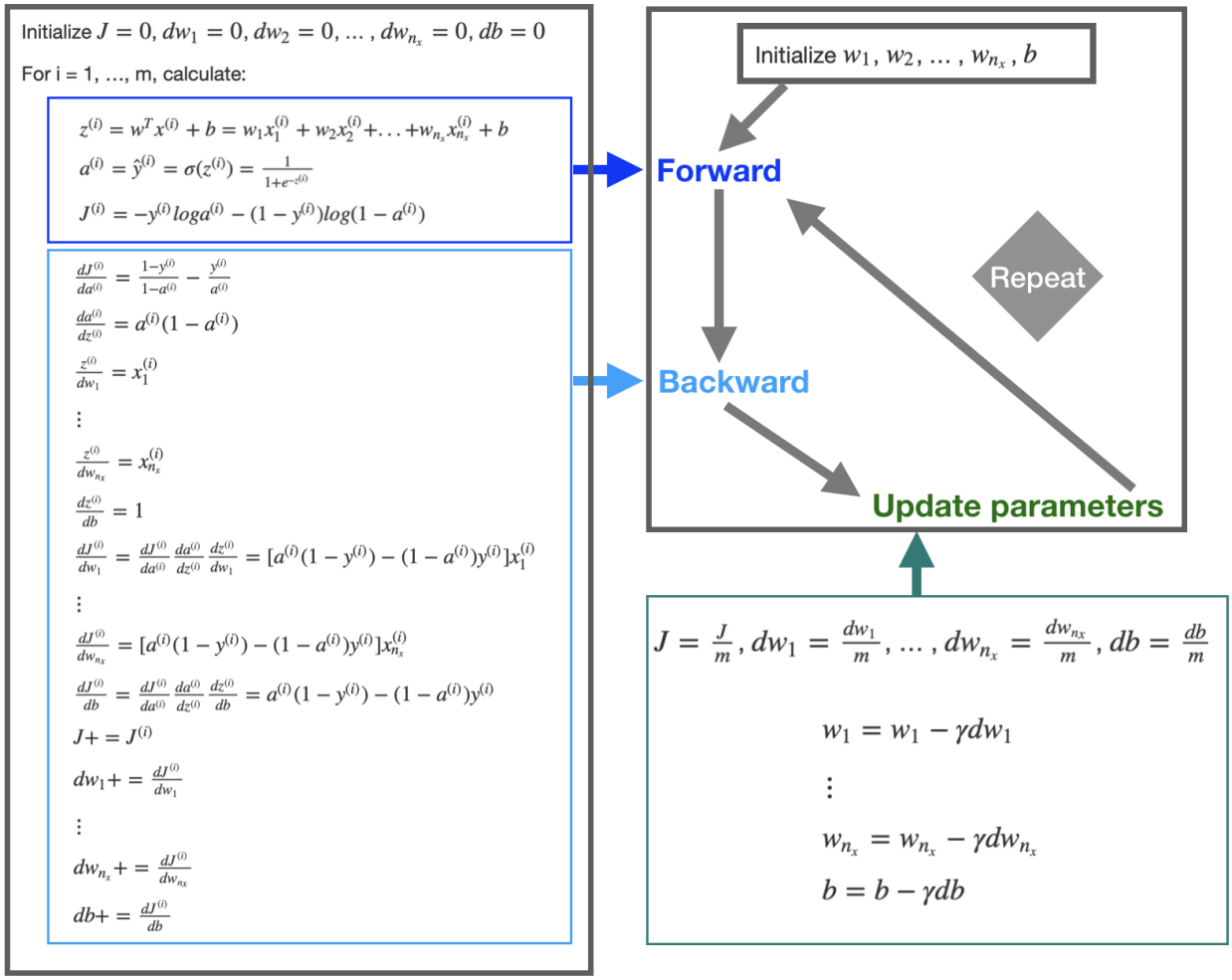
1.1.2 Stochastic Gradient Descent

The general approach to minimize $J(w,b)$ is by gradient descent, also known as *back-propagation*. The optimization process is a backward and forward sweep over the network.



The forward propagation takes the current weights and computes the prediction and cost. The backward propagation calculates the gradient descent for the parameters by the chain rule for differentiation. In logistic regression, it is easy to compute the gradient w.r.t the parameters (w,b).

The Stochastic Gradient Descent (SGD) for logistic regression across m samples. SGD updates one instance each time. The precise procedure is as follows.



First initialize w_1, w_2, ... , w_{n_x}, and b. Then fill in the initialized value to the forward and backward propagation. The forward propagation takes the current weights and estimates the prediction $\hat{h}^{(i)}$ and cost $J^{(i)}$. The backward propagation computes the gradient descent for the parameters. After repeating through all m samples, you can figure the gradient descent for the parameters. Then update the parameter by:

$$w := w - \gamma \frac{\partial J}{\partial w}$$
$$b := b - \gamma \frac{\partial J}{\partial b}$$

Replicate the forward and backward operation using the updated parameter until the cost J stabilizes

1.1.3 Deep Neural Network

Before the term deep learning was established, a neural network referred to a single hidden layer network. Neural networks with more than one layer are called deep learning. Network with the structure in figure 1.1 is the **multiple layer perceptron (MLP)** or **feedforward neural network (FFNN)**.

Let's look at a simple one-hidden-layer neural network (figure 1.2). First, only consider one sample. From left to right, there is an input layer with three features (x_1, x_2, x_3), a hidden layer with four neurons, and output later to produce a prediction $\hat{y}$.

From input to the first hidden layer Each inactivated neuron on the first layer is a linear transformation of the input vector x. For example, $z_1^{[1]} = w_1^{[1]T} x^{(i)} + b_1^{[1]}$ is the first inactivated neuron for hidden layer one. **We use superscript '[l]' to denote an amount associated with the l^{th} layer and the subscript 'i' to denote the i^{th} entry of a vector (a neuron or feature).** Here $w^{[1]}$ and $b_1^{[1]}$ are the weight and bias

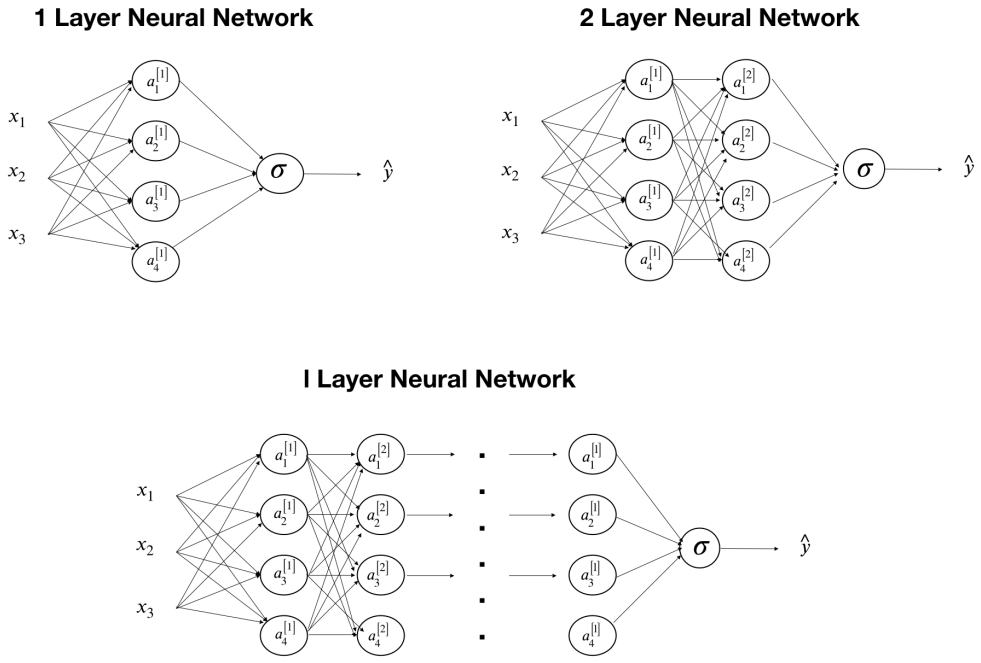


Figure 1.1 Feedforward Neural Network

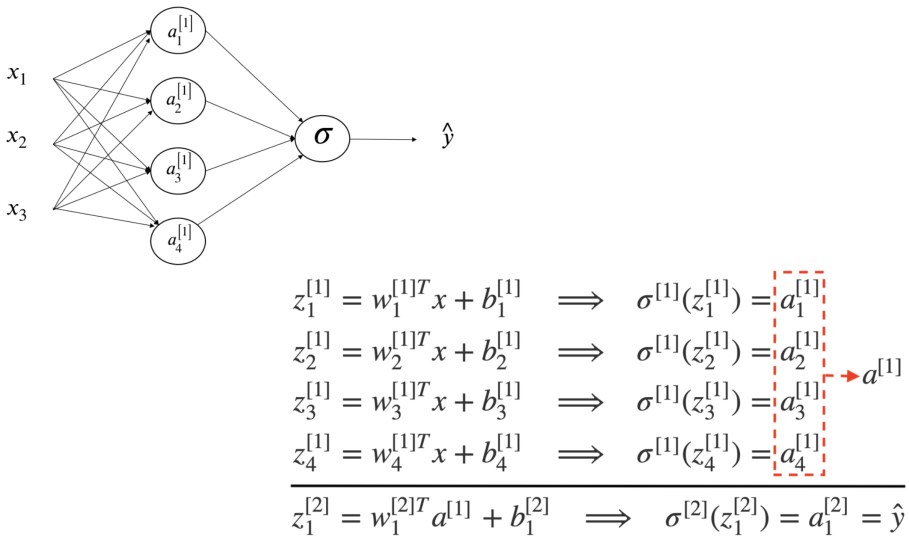


Figure 1.2 1-layer Neural Network

parameters for layer 1. $w^{[1]}$ is a 4×1 vector and hence $w_1^{[1]T} x^{(i)}$ us a linear combination of the four input features. Then use a sigmoid function $\sigma(\cdot)$ to activate the neuron $z_1^{[1]}$ to get $a_1^{[1]}$.

From the first hidden layer to the output Next, do a linear combination of the activated neurons from the first layer to get inactivated output, $z_1^{[2]}$. And then activate the neuron to get the predicted output $\hat{y}$. The parameters to estimate in this step are $w^{[2]}$ and $b_1^{[2]}$.

If you entirely write out the process, it is the bottom right of figure 1.2. When you execute a neural network, you need to do a similar computation four times to get the activated neurons in the first hidden layer. Doing this with a 'for' loop is inefficient. So people vectorize the four equations. As a vector, take input and calculate the affiliated z and a. You can vectorize each step and get the following illustration:

$$\begin{array}{ll} z^{[1]} = W^{[1]} x + b^{[1]} & \sigma^{[1]}(z^{[1]}) = a^{[1]} \\ z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} & \sigma^{[2]}(z^{[2]}) = a^{[2]} = \hat{y} \end{array}$$

$b^{[1]}$ Is the column vector of the four bias parameters displayed overhead? For example, $z^{[1]}$ is a column vector of the four non-active neurons. When you apply an activation function to a matrix or vector, you use it element-wise. $W^{[1]}$ is the matrix by stacking the four row-vectors:

$$W^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix}$$

So if you have one sample, you can iterate through the above forward propagation process to calculate the output $\hat{y}$ for that sample. However, if you have m training samples, you need to repeat this process for each of the m samples. We use superscript '(i)' to denote a quantity associated with i^{th} sample. You need to do the exact computation for all m samples.

For i = 1 to m, do:

$$\begin{array}{ll} z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} & \sigma^{[1]}(z^{[1](i)}) = a^{[1](i)} \\ z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]} & \sigma^{[2]}(z^{[2](i)}) = a^{[2](i)} = \hat{y}^{(i)} \end{array}$$

Recollect that we defined the matrix X as equal to our training samples stacked up as column vectors in equation 1.1. We do a similar procedure here to stack vectors with the superscript (i) together across m samples. This way, the neural network calculates the outputs on all the samples at the same time:

$$\begin{array}{ll} Z^{[1]} = W^{[1]} X + b^{[1]} & \sigma^{[1]}(Z^{[1]}) = A^{[1]} \\ Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]} & \sigma^{[2]}(Z^{[2]}) = A^{[2]} = \hat{Y} \end{array}$$

where

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(1)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix},$$

$$A^{[l]} = \begin{bmatrix} | & | & & | \\ a^{[l](1)} & a^{[l](1)} & \cdots & a^{[l](m)} \\ | & | & & | \end{bmatrix}_{l=1 \text{ or } 2},$$

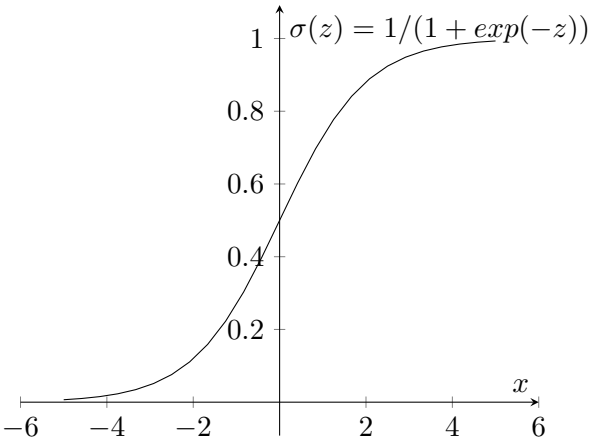
$$Z^{[l]} = \begin{bmatrix} | & | & & | \\ z^{[l](1)} & z^{[l](1)} & \cdots & z^{[l](m)} \\ | & | & & | \end{bmatrix}_{l=1 \text{ or } 2}$$

You can add layers like this to get a deeper neural network as shown in the bottom right of figure 1.1.

1.1.4 Activation Function

Sigmoid and Softmax Function

The sigmoid (or logistic) activation function. The function is S-shape with an output value between 0 to 1. Therefore it is used as the output layer activation function to forecast the probability **when the response y is binary**. However, it is infrequently used as a middle layer activation function. One of the main reasons is that when z is away from 0, the function's derivative drops fast and slows down the optimization process through gradient descent. Even though it is differentiable and provides some convenience, the decreasing slope can cause a neural network to get stuck at the training time.



People use the softmax function as the output layer activation function when the output has more than two categories.

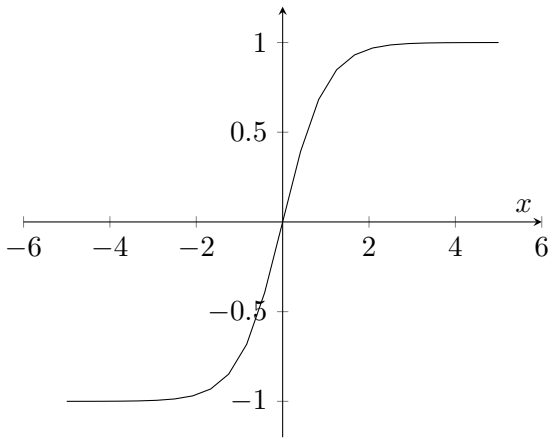
$$f_i(\mathbf{z}) = \frac{e^{z_i}}{\Sigma_{j=1}^{J} e^{z_j}} \tag{1.3}$$

where $\mathbf{z}$ is a vector.

Hyperbolic Tangent Function (tanh)

Another activation function with a similar S-shape is the hyperbolic tangent function. It often works better than the sigmoid function as the intermediate layer.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{1.4}$$



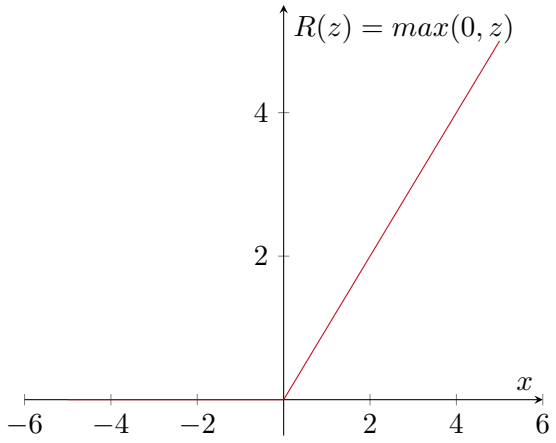
The tanh function crosses point (0, 0), and the value of the function is between 1 and -1, which makes the mean of the activated neurons closer to 0. The sigmoid function doesn't have that property. When you preprocess the training input data, you sometimes center the data so that the mean is 0. The tanh function does that data processing in some way, making learning for the next layer a little easier . This activation function is used a lot in the recurrent neural networks where you want to polarize the results.

Rectified Linear Unit (ReLU) Function

The Rectified Linear Unit (ReLU) is the most popular activation function. It is a piecewise function or a half rectified function:

$$R(z) = max(0, z) \tag{1.5}$$

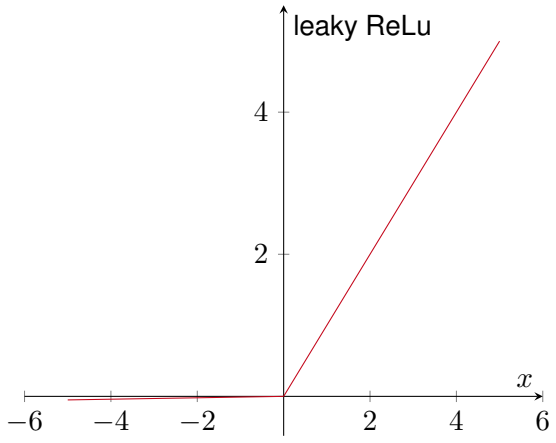
The derivative is 1 when z is positive and 0 when z is negative. You can define the derivative as either 0 or 1 when z is 0. When you implement this, it is unlikely that z equals precisely 0, even though it can be very close to 0.



The benefit of the ReLU is that when z is positive, the derivative doesn't vanish as z gets larger. So it leads to faster estimation than sigmoid or tanh. It is non-linear with an unconstrained response. However, the weakness is that when z is negative, the derivative is 0. It may not map the negative values correctly. In practice, this doesn't cause too much trouble, but there is another version of ReLu called leaky ReLu that attempts to solve the dying ReLU problem. The leaky ReLu is

$$R(z)_{Leaky} = \begin{cases} z & z \geq 0 \\ az & z < 0 \end{cases}$$

Instead of being 0 when z is negative, it adds a slight slope such as $a = 0.01$ as shown in figure **??** (can you see the leaky part there? :).



You may notice that all activation functions are non-linear. Since the arrangement of two linear functions is still linear, using a linear activation function doesn't help to apprehend more information. That is why you don't see people use a linear activation function in the middle layer. One exception is when the output y is continuous, you may use a linear activation function at the output layer. To sum up, for the middle layers:

· ReLU is usually a good choice. If you don't know what to choose, then start with ReLU. Leaky ReLu usually works better than the ReLU, but it is not used as much in practice. Either one works fine. Also, people usually use a = 0.01 as the slope for leaky ReLU. You can try different parameters, but most people a = 0.01.

· tanh is sometimes used, especially in a recurrent neural network. But you nearly never see people use the sigmoid function as a middle layer activation function.

For the output layer:

· When it is binary classification, use sigmoid with binary cross-entropy as a loss function

· When there are multiple classes, use the softmax function with categorical cross-entropy as a loss function

· When the response is continuous, use identity function (i.e. y = x), Which we are using in this case to do the regression on the energy dataset.

2 Method

2.1 Neural Network From Scratch

Listing 2.1 Sigmoid Implementation

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

def sigmoid_deriv(x):
    return sigmoid(x)*(1-sigmoid(x))
```

Listing 2.2 Forward propagation

```
def fwd(x, W_hid, W_out):
    assert x.ndim == 2
    assert W_hid.ndim == 2
    assert W_out.ndim == 2

    z_hid = x @ W_hid
    h_hid = sigmoid(z_hid)                    # hidden output

    z_out = h_hid @ W_out
    y_hat = z_out                             # linear output

    return y_hat, z_hid, h_hid                # z_hid and h_hid required for backprop
```

Listing 2.3 Back propagation

```
def backprop(x, y, W_hid, W_out):
    assert x.ndim == 2
    assert y.ndim == 2
    assert W_hid.ndim == 2
    assert W_out.ndim == 2

    y_hat, z_hid, h_hid = fwd(x, W_hid, W_out)

    ro_out = -(y-y_hat)                                  # no transfer function
    dW_out = h_hid.T @ ro_out

    ro_hid = (ro_out @ W_out.T) * sigmoid_deriv(z_hid)
    dW_hid = x.T @ ro_hid

    return dW_hid, dW_out, y_hat
```

Listing 2.4 Hyperparameter

```
num_layer =1
n_in = 8
n_hid1 = 4
n_hid = 4
n_out = 1
lr = 0.01
n_batch = 800
iterations = 3000
```

Listing 2.5 Initialize weights

```
# Initialize weights
W_hid = np.random.normal(0.0, n_in**-.5, [n_in, n_hid])
W_out = np.random.normal(0.0, n_hid**-.5, [n_hid, n_out])
```

Listing 2.6 Training loop in batches

```
losses = {'train': [], 'valid': []}  # keep history for plotting

for i in range(iterations):

    # Get mini-batch, both as 2d arrays
    batch = np.random.choice(train_i, n_batch)
    x = train_x[batch]
    y = train_y[batch]


    dW_hid, dW_out, y_hat = backprop(x, y, W_hid, W_out)
    W_hid += -lr * dW_hid / n_batch                      # Apply gradient
    W_out += -lr * dW_out / n_batch
    # Keep train loss (on mini-bNatch) and calculate validation loss
    train_loss = MSE(y, y_hat)
    losses['train'].append(train_loss)
    train_y_hat, _, _ = fwd(train_x, W_hid, W_out)
    valid_y_hat, _, _ = fwd(valid_x, W_hid, W_out)
    valid_loss = MSE(valid_y, valid_y_hat)
    losses['valid'].append(valid_loss)
    test_y_hat, _, _ = fwd(test_x, W_hid, W_out)
```

2.2 Keras Neural Network

Let's look at how to use the **keras** package for example in deep learning with the Building energy dataset .

The following code was used to validate the results obtained by the neural network we created from scratch.

Listing 2.7 Some Code

```
# Create your first MLP in Keras
# imports

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn.metrics import r2_score
import matplotlib.pyplot as plt
import numpy
from tensorflow.keras.optimizers import Adam
import keras
from matplotlib import pyplot
from keras.callbacks import EarlyStopping
import pandas as pd
from sklearn.preprocessing import LabelEncoder
# Read data from csv file for training and validation data
# Split into input (X) and output (Y) variables
X1 = train_x
Y1 = train_y

X2 = valid_x
Y2 = valid_y
# Create model
model = Sequential()
model.add(Dense(4, activation="relu", input_dim=8))

# Since the regression is performed, a Dense layer containing a single neuron with a linear activation function.
# Typically ReLu-based activation are used but since it is performed regression, it is needed a linear activation.
model.add(Dense(1, activation="linear"))

# Compile model: The model is initialized.
model.compile(loss='mean_squared_error', optimizer=Adam(lr=0.07, decay=1e-3 / 200))

# Patient early stopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=200)

# Fit the model
history = model.fit(X1, Y1, validation_data=(X2, Y2), epochs=3000, batch_size=128, verbose=2,
```

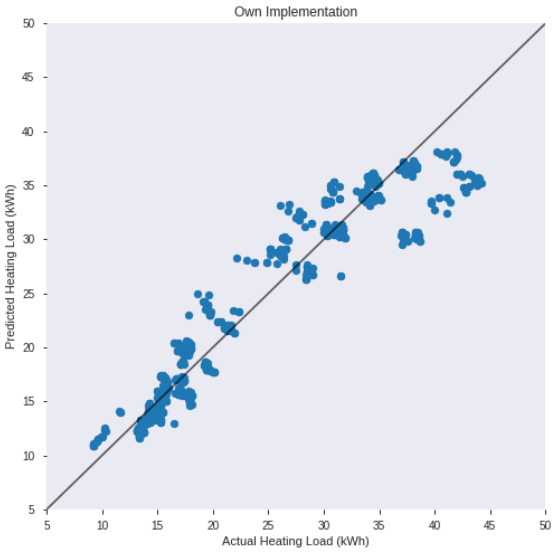
Please Refer to the included Jupyter notebook, to run and further details

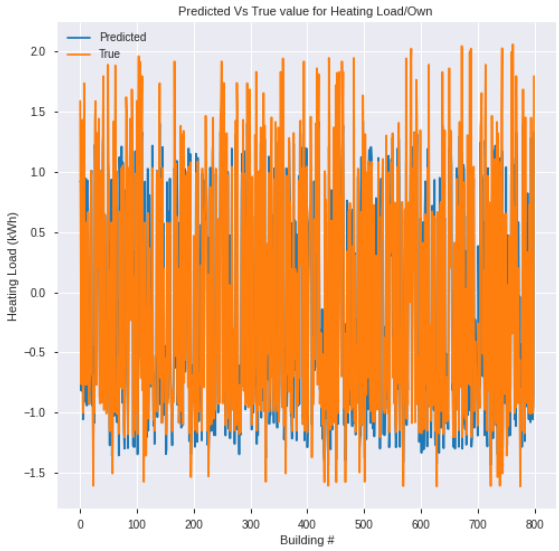
3 Results

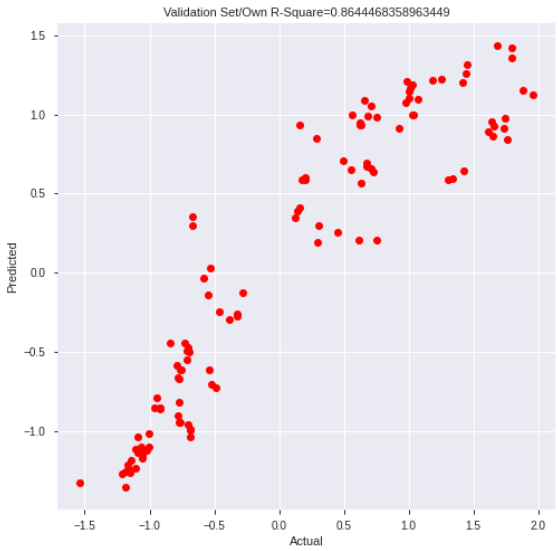
To compare the two methods and validate the correctness of the self-implemented neural network. We used Keras, a widely used package with the same number of layers and metrics, and we can see from Figures 3.1 and 3.2 that the results are similar. When using the Keras, it was made sure that we were not splitting the data again but rather using the same training and testing dataset for both methods to ensure consistency.

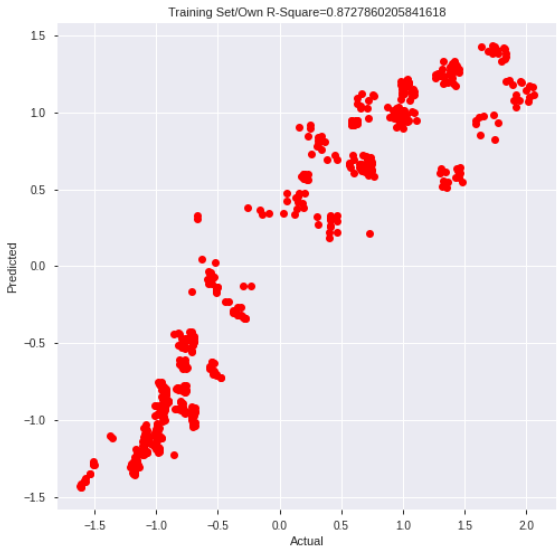
Appendix A.2 shows how to implement a simple two-layer neural network and train it on the MNIST digit recognizer dataset. It's meant to be an instructional example. You can better understand the underlying math of neural networks. The NN has a simple two-layer architecture. For example, input layer a[0] will have 784 units corresponding to the 784 pixels in each 28x28 input image.

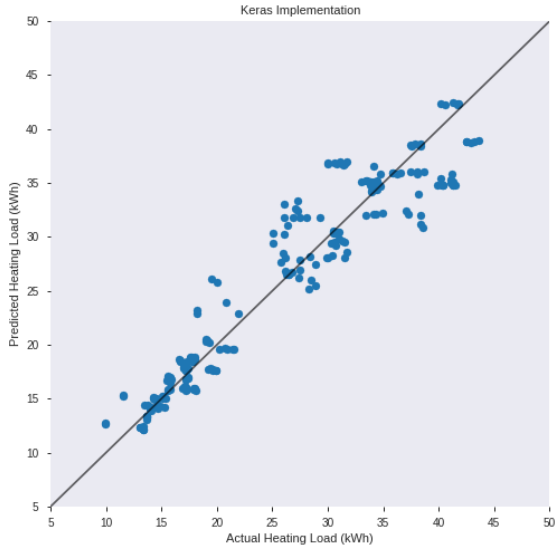
In Appendix A.2, the beauty of this example and the dataset is how visual it can be, and imlementing it from scratch was challenging. Still, in the end, it depends on my understanding and always gives visual feedback.

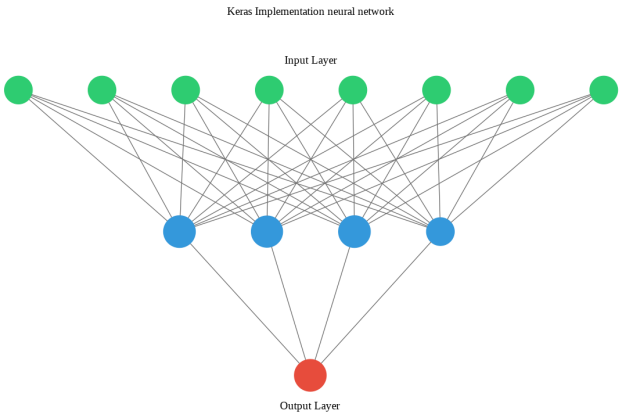


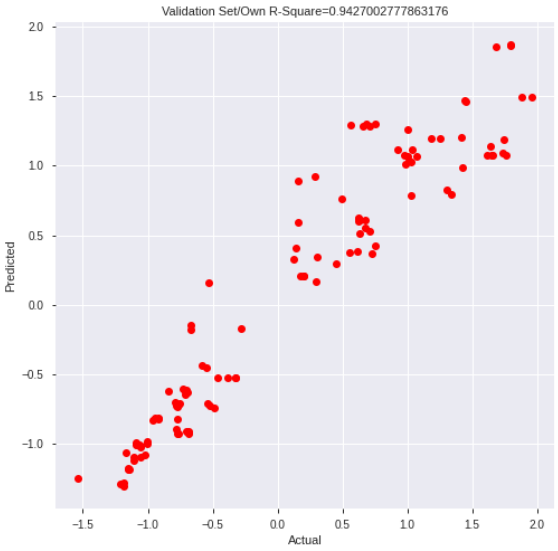


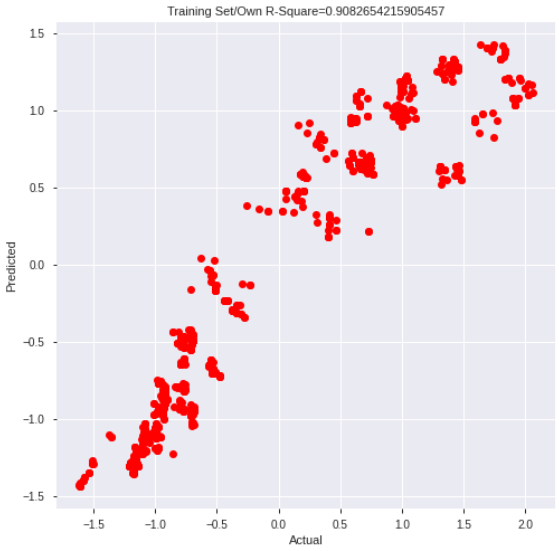












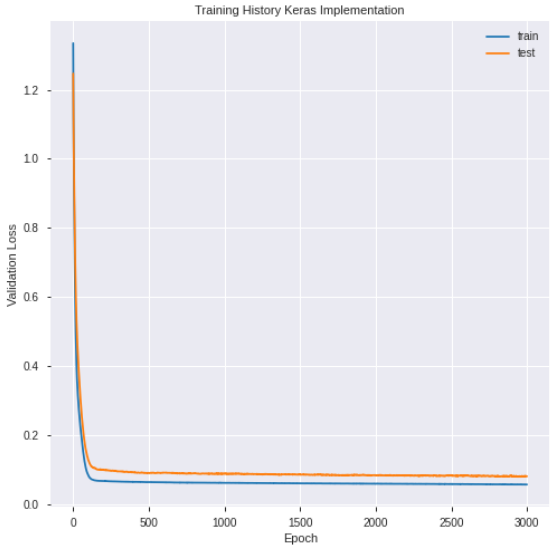


Figure 3.2 Keras Neural Network Results

However, Figure 3.1 shows that the self-made model has 92 % training and 88% validation accuracy. This means that the model performs well on both data sets, thus indicating no overfitting or underfitting.

For the Keras Neural Network, the model has 94 % training and 92% validation accuracy. Although we used a similar parameter for both models, the Keras model performed slightly better. After digging, some online Forums suggested that Keras is much more optimized in dealing with Floating-point arithmetic.

Overall, the results confirmed my understanding of the inner working of the Neural network.

4 Conclusions

In summary, we have achieved the following in this project:

· Objectives set in the project proposal and presentation were met.

· Both Neural Networks executed successfully without diverging in most cases.

· Produces reasonable results, even for relatively complex dataset

However, we also observed the following shortcomings in the project:

· Adding more layers to the self-implemented NN proved to be more challenging than initially thought.

· Using different versions of Keras have its issues, and multiple hours were spent on fixing dependency issues.

In retrospect, we can also make the following observations:

· The goals and objectives in the project proposal were reasonable for the project's timeframe.

· Making my implementation and using the MNIST digit data set was more beneficial than the data provided by the course.

Future projects may also wish to explore the following extensions:

· Extension to automatically add more hidden layers in the self-implementation

· Hyperparameter optimization "This by itself is a science."

· Implementing Adaptive Moment Estimation (Adam) from scratch would be interesting, and

· Comparing it to other Optimization techniques

A Appendix

A.1 Calculation example of Feed Forward Neural Network

Let's consider the calculation of a single unit of a neural network. z is the weighted input, y is the output, and (z) is the activation function representing a sigmoid function. Nowadays, since the sigmoid function causes "The vanishing gradient problem," the ReLU function as an activation function is recommended. However, the sigmoid function was a standard differentiable activation function that substituted the step function, the function of imitating neurons inside a human's brain. In the following tutorial, we are going to use the sigmoid function so that we can deepen our understanding of the back-propagation. Therefore, in this section, we will use the sigmoid function.

$$z = x_1 w_1 + x_2 w_2 + b$$
$$y = (z) = \frac{1}{1+\exp(-z)}$$

x1, x2 are the inputs. w1 and w2 are the coefficient weights for each piece of information. x1 and x2 are data that have been processed with normalization or standardization. Methods such as applying normalization or standardization to input data allow better performance. For example, when normalizing a picture with a range of 0 255, we divide the picture by 255, thus obtaining a color range of 0 to 1. The weights initialize in a small field, so gradients exist in the initial learning state. There are methods such as Gaussian distribution usage to initialize weights. In this tutorial, we will set the values from 0 to 1. Bias is set to 0 due to the initial 0 bias performing better learning accuracy. As learning goes by, the bias will also be updated.

The following are the computation examples of using random input and weight values ranging in 0 1, with bias as 0:

x1	x2	w1	w2	weighted input z	sigmoid	y
0.1	0.3	0.6	0.2	$z = 0.1 * 0.6 + 0.3 * 0.2 + 0$	$\sigma(z) = \frac{1}{1+\exp(-0.12)} = 0.530$	0.530
0.7	0.2	0.8	0.1	$z = 0.7 * 0.8 + 0.2 * 0.1 + 0$	$\sigma(z) = \frac{1}{1+\exp(-0.58)} = 0.641$	0.641
0.2	0.4	0.9	0.5	$z = 0.2 * 0.9 + 0.4 * 0.5 + 0$	$\sigma(z) = \frac{1}{1+\exp(-0.38)} = 0.594$	0.594

Figure A.1 The intuition behind the weighted average gradient

Variables and Parameters

The figure above describes the neural network classifying a 4×3 pixel picture.

Here, we will introduce variables and parameters commonly used in thesis or interpretations. The figure would help you grasp it more deeply.

x_i^{l-1} : The output from unit i in layer l-1. It becomes the inputs of the units in layer l.

x_j^l: The output from unit j in layer l. It becomes the inputs of the units in layer l+1.

w_{ji}^l: The weight bears on the signal which is outputted from unit i in layer l-1 and goes into unit j in layer l.

b_j^l: The bias bears on unit j in layer l.

z_j^l: The weighted input of unit j in layer l.

$_j^l(z_j^l)$: The activation function of unit j in layer l

(A.1)

The following are examples:

x_2^1 : The output from unit 2 in layer 1. It becomes the inputs of the units in layer 2.

x_3^2: The output from unit 3 in layer 2. It becomes the inputs of the units in layer 3.

w_{31}^2: The weight bears on the signal which is outputted from the unit No. 1 in layer one and goes into unit 3 in layer 2.

b_3^2: The bias bears on unit 3 in layer 2.

z_3^2: The weighted input of unit 3 in layer 2.

$_3^2(z_3^2)$: The activation function of unit 3 in layer 2

(A.2)

Binary Classification of a Picture Using 1 Unit

We will consider a neural network that classifies whether the number written in the picture is one or not. For this type of problem, we use the sigmoid function as output. The rest of the units use the sigmoid function for a reason mentioned in the previous section.

Let's focus on a particular unit and calculate it. If we assume input pictures are described in black-and-white, then inputs of the input layer (layer 1) can be expressed as follows:

$$x_i^1 = \begin{cases} 0 \\ 1 \end{cases}$$

Since the activation function of the input layer is an identity function(y=x), the inputs are calculated with no changes and become the inputs for the hidden layer (layer 2). For example, let's say we put the following picture as input:

1	2	3
4	5	6
7	8	9
10	11	12

Figure A.2 The intuition behind the weighted average gradient

Doing so, the inputs of the input layer, that is, the information of the hidden layer would be as follows:

$$\begin{array}{lll} x_1^1 = 1, & x_2^1 = 1, & x_3^1 = 0 \\ x_4^1 = 0, & x_5^1 = 1, & x_6^1 = 0 \\ x_7^1 = 0, & x_8^1 = 1, & x_9^1 = 0 \\ x_{10}^1 = 0, & x_{11}^1 = 1, & x_{12}^1 = 0 \end{array}$$

Let's see how the calculation will go in unit 1 of layer 2.
As we did in section 1, the weights are set as random (0 1 value), and the bias is set as 0.

$$\begin{array}{l} w_{11}^2 = 0.48,\ w_{12}^2 = 0.29,\ w_{13}^2 = 0.58,\ w_{14}^2 = 0.61,\ w_{15}^2 = 0.47,\ w_{16}^2 = 0.18, \\ w_{17}^2 = 0.02,\ w_{18}^2 = 0.49,\ w_{19}^2 = 0.67,\ w_{1\ 10}^2 = 0.62,\ w_{1\ 11}^2 = 0.017,\ w_{1\ 12}^2 = 0.28 \\ b_1^2 = 0 \end{array}$$

The weighted input would be as follows:

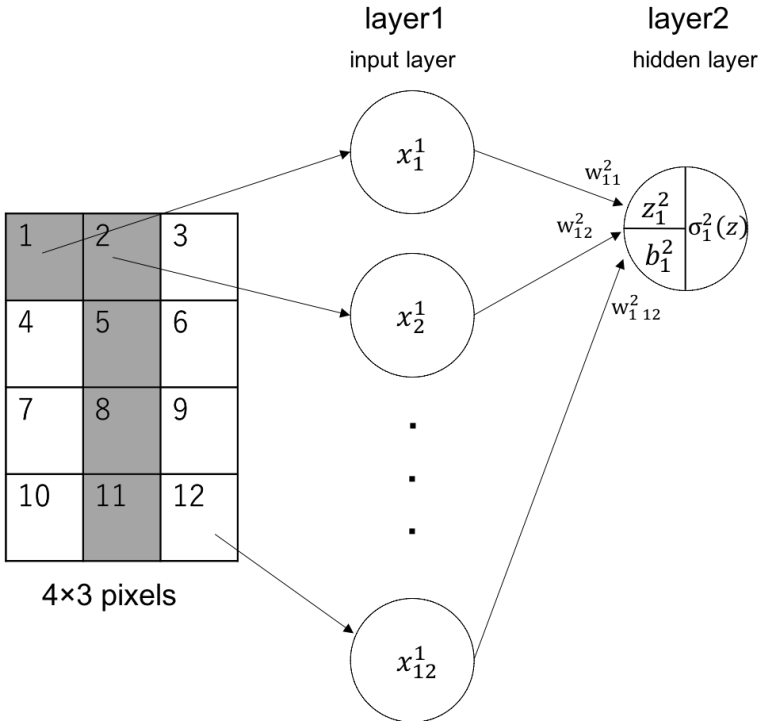


Figure A.3 The intuition behind the weighted average gradient

$$
\begin{aligned}
& z_1^2 \\
=\ & x_1^1 w_{11}^2 + x_2^1 w_{12}^2 + x_3^1 w_{13}^2 + x_4^1 w_{14}^2 + x_5^1 w_{15}^2 + x_6^1 w_{16}^2 + x_7^1 w_{17}^2 \\
& + x_8^1 w_{18}^2 + x_9^1 w_{19}^2 + x_{10}^1 w_{1\,10}^2 + x_{11}^1 w_{1\,11}^2 + x_{12}^1 w_{1\,12}^2 + b_1^2 \\
=\ & 1\ddot{O} w_{11}^2 + 1\ddot{O} w_{12}^2 + 0\ddot{O} w_{13}^2 + 0\ddot{O} w_{14}^2 + 1\ddot{O} w_{15}^2 + 0\ddot{O} w_{16}^2 + 0\ddot{O} w_{17}^2 \\
& + 1\ddot{O} w_{18}^2 + 0\ddot{O} w_{19}^2 + 0\ddot{O} w_{1\,10}^2 + 1\ddot{O} w_{1\,11}^2 + 0\ddot{O} w_{1\,12}^2 + b_1^2 \\
=\ & 1\ddot{O}0.48 + 1\ddot{O}0.29 + 0\ddot{O}0.58 + 0\ddot{O}0.61 + 1\ddot{O}0.47 + 0\ddot{O}0.18 + 0\ddot{O}0.02 \\
& + 1\ddot{O}0.49 + 0\ddot{O}0.67 + 0\ddot{O}0.62 + 1\ddot{O}0.017 + 0\ddot{O}0.28 + 0 \\
=\ & 1\ddot{O}0.48 + 1\ddot{O}0.29 + 1\ddot{O}0.47 + 1\ddot{O}0.49 + 1\ddot{O}0.017 + 0 \\
=\ & 1.277
\end{aligned}
\tag{A.3}
$$

We insert the value into an activation function. We will use the sigmoid function for the same reason in section 1. The output of the unit will be,

$$
\begin{aligned}
x_1^2 =_1^2 (z_1^2) &= \tfrac{1}{1+\exp(-z_1^2)} \\
&= \tfrac{1}{1+\exp(-(1.277))} \\
&= 0.782
\end{aligned}
$$

The exact computations are conducted in other units of the neural network. In the following tutorial ("Calculation example of Feed Forward Neural Network (2)"), we will calculate all of the units in the neural network and see how the final output turns out.

A.2 Two-layer neural network from Scratch to recognize MNIST digit data set

In this notebook, I implemented a simple two-layer neural network and trained it on the MNIST digit recognizer dataset. It's meant to be an instructional example, through which you can understand the underlying math of neural networks better.

Our NN will have a simple two-layer architecture. Input layer $a^{[0]}$ will have 784 units corresponding to the 784 pixels in each 28x28 input image. A hidden layer $a^{[1]}$ will have 10 units with ReLU activation, and finally our output layer $a^{[2]}$ will have 10 units corresponding to the ten digit classes with softmax activation.

Forward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}))$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$$

Backward propagation

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m} \Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m} \Sigma dZ^{[1]}$$

Parameter updates

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

Vars and shapes
Forward prop

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]}A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]}A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

Please Refer to the Jupyter notebook for further details

A.3 Optimization

We need to determine how to update the network based on a loss function (a.k.a. objective function). In this section, we will look at variants of optimization algorithms that will enhance the training speed.

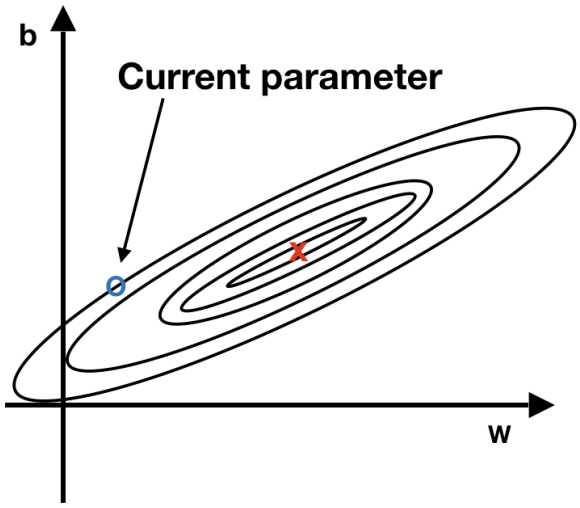


Figure A.4 The intuition behind the weighted average gradient

A.3.1 Batch, Mini-batch, Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) revises one sample each time. The SGD for logistic regression across some sample m. If you process the whole training set each time to calculate the gradient, it is **Batch Gradient Descent (BGD)**.

The training set is often monumental in deep learning applications, with hundreds of thousands or millions of samples. It could be slow if processing all the samples can only lead to one step of gradient descent. An intelligent method to improve the algorithm is to make some progress before finishing the entire data set. In particular, split the training set into smaller subsets and fit the model using one subset each time. The subsets are mini-batches. **Mini-batch Gradient Descent (MGD)** is to split the training set into smaller batches. For example, if the smaller batch size is 1000, the algorithm will process 1000 samples each time, compute the gradient and update the parameters. Then it moves on to the next batch group until it goes through the entire training set. We call it one pass-through training set using mini-batch gradient descent or one Epoch. **Epoch** is a common keyword in deep learning, which means a single pass through the training set. For example, if the training batch has 60,000 samples, one Epoch leads to 60 gradient descent steps. Then it will start over and take another pass via the training set. It means one more find to make, the optimal number of epochs. It is decided by looking at the trends of execution metrics on a holdout set of training data. We examined the data splitting and sampling in Jupyter notebook. Usually, a single holdout is set to tune the model in deep learning. But it is essential to use a big enough holdout set to give high confidence in your model's overall performance. Then you can estimate the selected model on your test set that is not used in the training process. MGD is what everyone in deep learning uses when training on an extensive data set.

$$\underset{(n_x,m)}{x =} \quad [\underbrace{x^{(1)}, x^{(2)}, \cdots, x^{(1000)}}_{mini-batch\ 1} / \quad \cdots / \cdots x^{(m)}]$$

$$\underset{(1,m)}{y =} \quad [\underbrace{y^{(1)}, y^{(2)}, \cdots, y^{(1000)}}_{mini-batch\ 1} / \quad \cdots / \cdots y^{(m)}]$$

- Mini-batch size = m: batch gradient descent, too long per iteration - Mini-batch size = 1: stochastic gradient descent, loss speed from vectorization - Mini-batch size in between: mini-batch gradient descent, make progress without processing all training set, typical batch sizes are $2^6 = 64$, $2^7 = 128$, $2^7 = 256$, $2^8 = 512$

A.3.2 Optimization Algorithms

In the history of deep learning, researchers suggested other optimization algorithms and demonstrated that they performed well in a specific scenario. But the optimization algorithms didn't generalize well to a wide range of neural networks. So you will need to try different optimizers in your application. The following introduces three typically used optimizers here, and they are all based on something called exponentially weighted averages. To comprehend its intuition, let's look at a theoretical example shown in figure A.4.

We have two parameters, b, and w. The blue dot denotes the current parameter value, and the red point is the optimum value we want to reach. The current value is close to the target vertically but far out horizontally. In this position, we hope to have slower learning on b and faster learning on w. A way to alter this is to use the average of the gradients from other iterations instead of the current iteration to update the parameter. Vertically, since the current value is close to the target value of parameter b, the gradient of b is likely to bounce between positive and negative values. The average manages to cancel out the positive and negative derivatives along the vertical direction and slow down the oscillations. Horizontally, since it is still further out from the target value of parameter w, all the gradients will likely indicate the same direction. So using an average won't have too much influence there. That is the basic idea behind many optimizers: adjust the learning rate using a weighted average of different iterations' gradients.

Exponentially Weighted Averages

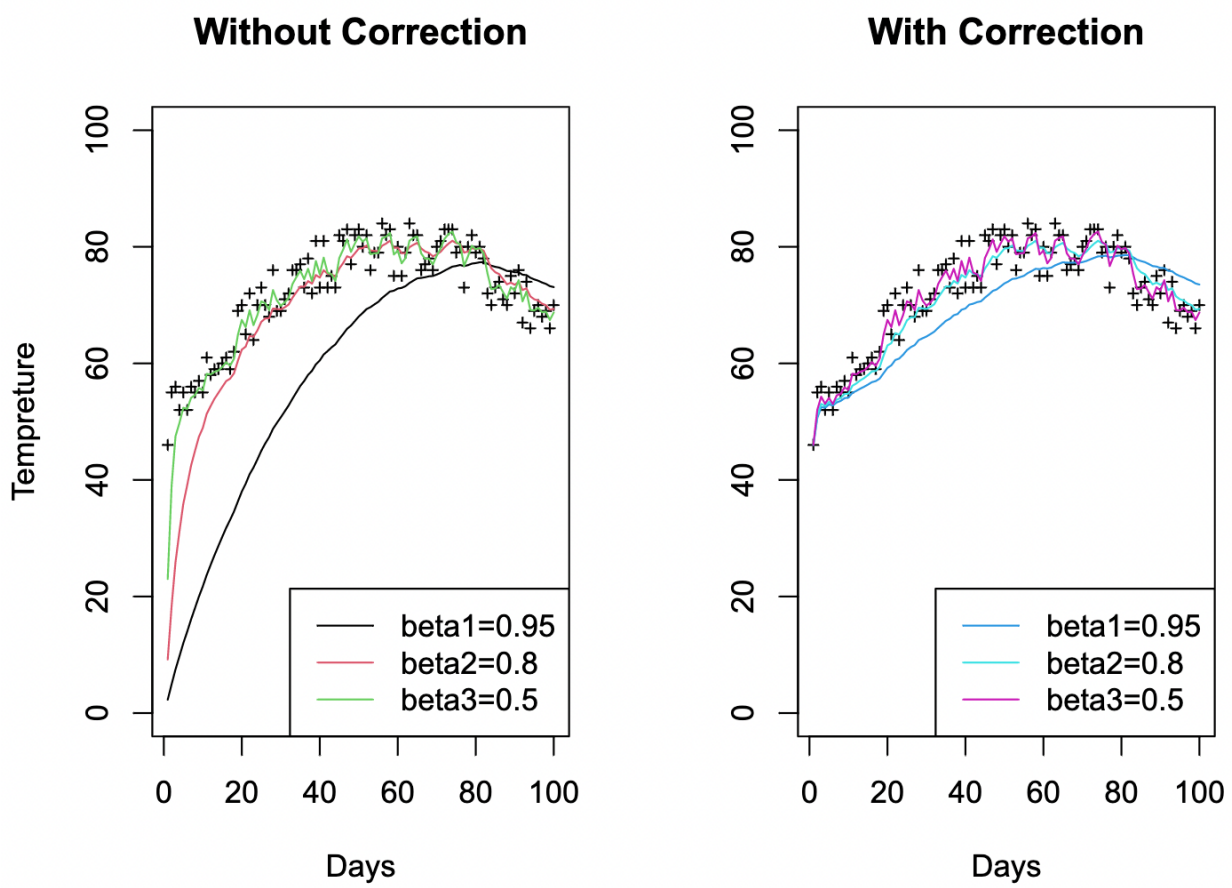


Figure A.5 Exponentially weighted averages with and without correction

Before we get to more sophisticated optimization algorithms that implement this idea, let's first introduce the basic weighted moving average framework(**?**).

Suppose we have the following 100 days' temperature data:

$$\theta_1 = 49F, \theta_2 = 53F, \ldots, \theta_{99} = 70F, \theta_{100} = 69F$$

The weighted average is defined as:

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

And we have:

$$\begin{gathered} V_0 = 0 \\ V_1 = \beta V_1 + (1-\beta)\theta_1 \\ V_2 = \beta V_1 + (1-\beta)\theta_2 \\ \vdots \\ V_{100} = \beta V_{99} + (1-\beta)\theta_{100} \end{gathered}$$

For example, for $\beta = 0.95$:

$$\begin{gathered} V_0 = 0 \\ V_1 = 0.05\theta_1 \qquad \ldots\ldots \\ V_2 = 0.0475\theta_1 + 0.05\theta_2 \end{gathered}$$

The black line in the right plot of figure A.5 is the exponentially weighted averages of simulated temperature data with $\beta = 0.95$. V_t is around average over the prior $\frac{1}{1-\beta}$ days. So $\beta = 0.95$ resembles a 20 days' average. The red line corresponds to $\beta = 0.8$, which resembles five days average. As β increases, it averages over a larger window of the previous values; therefore, the curve gets smoother. A larger β also means that it gives the current value θ_t less weight $(1-\beta)$, and the average adjusts more slowly. It is straightforward to see from the plot that the averages at the beginning are more biased. The bias modification can help to achieve a better estimation:

$$V_t^{corrected} = \frac{V_t}{1-\beta^t}$$

$$V_1^{corrected} = \frac{V_1}{1-0.95} = \theta_1$$

$$V_2^{corrected} = \frac{V_2}{1-0.95^2} = 0.4872\theta_1 + 0.5128\theta_2$$

For $\beta = 0.95$, the original $V_2 = 0.0475\theta_1 + 0.05\theta_2$ which is a small fraction of both $theta_1$ and $theta_2$. That is why it starts so much lower with a gigantic bias. After correction, $V_2^{corrected} = 0.4872\theta_1 + 0.5128\theta_2$ is a weighted average with two weights added up to 1 which dismisses the bias. See that as t increases, β^t converges to 0 and $V^{corrected}$ converges to V^t.

How do we use the exponentially weighted average for optimization? Rather than using the gradients (dw and db) to update the parameter, we use the gradients' exponentially weighted average. There are different optimizers built on top of this idea. We will look at three optimizers, Momentum, Root Mean Square Propagation (RMSprop), and Adaptive Moment Estimation (Adam).

Momentum

The momentum algorithm uses the average of the exponentially weighted gradient to update the parameters. For example, on iteration t, calculate dw and db using samples in one mini-batch and update the parameters as follows:

$$V_{dw} = \beta V_{dw} + (1-\beta)dw$$

$$V_{db} = \beta V_{db} + (1-\beta)db$$

$$w = w - \alpha V_{dw};\ \ b = b - \alpha V_{db}$$

The learning rate α and weighted average parameter β are hyperparameters. The most common and strong choice is $\beta = 0.9$. This algorithm does not use the bias correction because the average will warm up after a dozen iterations and no longer be biased(**?**). The momentum algorithm typically works better than the actual gradient descent without any average.

Root Mean Square Propagation (RMSprop)

The Root Mean Square Propagation (RMSprop) is another algorithm that uses the idea of an exponentially weighted average. On iteration t, compute dw and db using the current mini-batch(**?**). Rather than V, it computes the weighted average of the squared gradient descents(**?**). When dw and db are vectors, the squaring is an element-wise operation.

$$S_{dw} = \beta S_{dw} + (1-\beta)dw^2$$

$$S_{db} = \beta S_{db} + (1-\beta)db^2$$

Then, update the parameters as follows:

$$w = w - \alpha\frac{dw}{\sqrt{S_{dw}}};\ b = b - \alpha\frac{db}{\sqrt{S_{db}}}$$

The RMSprop algorithm splits the learning rate for a parameter by a weighted average of recent gradients' importance for that parameter. The goal is still to modify the learning speed. For example, when parameter b is close to its target value, we want to reduce the oscillations along the vertical direction.

Adaptive Moment Estimation (Adam)

The Adaptive Moment Estimation (Adam) algorithm is, in some way, a combination of momentum and RMSprop. On iteration t, compute dw, db using the current mini-batch. Then calculate both V and S using the gradient descents.

$$\begin{cases} V_{dw} = \beta_1 V_{dw} + (1-\beta_1)dw \\ V_{db} = \beta_1 V_{db} + (1-\beta_1)db \end{cases} \quad momantum\ update\ \beta_1$$

$$\begin{cases} S_{dw} = \beta_2 S_{dw} + (1-\beta_2)dw^2 \\ S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2 \end{cases} \quad RMSprop\ update\ \beta_2$$

The Adam algorithm implements bias correction.

$$\begin{cases} V_{dw}^{corrected} = \frac{V_{dw}}{1-\beta_1^t} \\ V_{db}^{corrected} = \frac{V_{db}}{1-\beta_1^t} \end{cases} ; \begin{cases} S_{dw}^{corrected} = \frac{S_{dw}}{1-\beta_2^t} \\ S_{db}^{corrected} = \frac{S_{db}}{1-\beta_2^t} \end{cases}$$

And it updates the parameter using both corrected V and S. ϵ here is a tiny positive number to assure the denominator is larger than zero. The choice of ϵ doesn't matter much, and the creators of the Adam algorithm suggested $\epsilon = 10^{-8}$. For hyperparameter β_1 and β_2, the common settings are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

$$w = w - \alpha\frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \epsilon};\ b = b - \alpha\frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon}$$

A.4 Deal with Overfitting

The most prominent trouble for deep learning is overfitting. It happens when the model learns too much from the data. A common way to diagnose the problem is cross-validation(**?**). You can identify the situation when the model fits excellent on the training data but gives poor forecasts on the testing data(**?**). One way to prevent the model from overlearning the data is to limit model sophistication. There are several methods to that.

Regularization

For logistic regression(**?**),

$$\underset{w,b}{min} J(w,b) = \frac{1}{m}\Sigma_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + penalty$$

Common penalties are L1 or L2 as follows:

$$L_2\ penalty = \frac{\lambda}{2m} \parallel w \parallel_2^2 = \frac{\lambda}{2m}\Sigma_{i=1}^{n_x} w_i^2$$

$$L_1\ penalty = \frac{\lambda}{m}\Sigma_{i=1}^{n_x} |w|$$

For a neural network,

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m}\Sigma_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2}\Sigma_{l=1}^{L} \parallel w^{[l]} \parallel_F^2$$

where

$$\parallel w^{[l]} \parallel_F^2 = \Sigma_{i=1}^{n^{[l]}}\Sigma_{j=1}^{n^{[l-1]}}(w_{ij}^{[l]})^2$$

They are usually called "Frobenius Norm" instead of L2-norm.

Dropout

Another robust regularization method is "dropout." For example, the random forest model de-correlates the trees by randomly choosing a subset of features. Dropout uses a similar idea in the parameter estimation process.

It temporally freezes a randomly chosen subset of nodes at a distinctive layer in the neural network during the optimization method to decrease overfitting. When using dropout to a respective layer and mini-batch, we pre-set a percentage, for example, 30%, and randomly dismiss 30% of the layer's nodes. The output from the 30% removed nodes will be zero. During backpropagation, we update the remaining parameters for a much-diminished network. Then, we need to do the random dropout for the next mini-batch.

To standardize the output with all nodes in this layer, we need to scale the work to the same percentage to ensure the dropped-out nodes do not affect the overall signal. The dropout process randomly turns off different nodes for each mini-batch. Dropout lessens overfitting in deep learning more efficiently than L1 or L2 regularizations(**?**).

Bibliography

[1] S. S. Haykin, *Neural networks and learning machines*, 3rd ed. Upper Saddle River, NJ: Pearson Education, 2009.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.