Week 4: Advances in neural networks

Introduction

In Week 4, you will continue your learning in the area of neural networks.

You will cover more advanced strategies to train neural networks such as momentum and stochastic-gradient descent this week. You will learn about regularisation techniques and dropouts to prevent overfitting. You will develop neural networks for regression and time series prediction problems. You will also use prominent libraries such as Keras and scikit-learn to train neural networks both in Python and R. Finally, you will be briefly introduced to deep learning with applications.

Recommended readings

We recommend you to **first go through the lessons and then see the related topics in the textbook**. Note that **the textbook readings are not mandatory.** They are meant to provide additional information for this week's lessons.

The following chapters of the course textbook will help you with the topics covered this week:

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd Ed.). O'Reilly Media, Inc.

- Chapter 10: Introduction to Artificial Neural Networks with Keras
- Chapter 12: Custom Models and Training with TensorFlow
- Chapter 13: Loading and Preprocessing Data with TensorFlow

Mitchell, T. (1997). Machine Learning. McGraw-Hill.

- Chapter 4: Neural Networks

All readings are available from the course Leganto reading list. Please keep in mind that you will need to be logged into Moodle to access the Leganto reading list.

This week's lessons were prepared by Dr. R. Chandra. If you have any questions or comments, please email directly: rohitash.chandra@unsw.edu.au

Momentum

Let's start this week by learning two approaches to speed up the training process involved in machine learning.

Momentum

Gradient descent employed by backpropagation essentially looks for the lowest peak of the door surface with steps taken in weight update determined by gradients at the particular points in weight space. This can take hundreds or thousands of epochs which is a problem with very large neural networks and big data problems. Hence, finding ways to improve the training process in terms of training with a lower number of epochs has been a challenge in machine learning.

The idea of using a momentum-based weight update is one of the strategies to speed up the training process. It is motivated by the concept of momentum used in the area of physics of bodies at motion. The idea is to utilise previous weight update in present weight update that acts as momentum for speeding the training process. The previous weight is multiplied to a constant similar to the learning rate and called momentum-rate. The below equations show how the change of weight is done in a momentum-based update.

$$\Delta w_{ij} = \left(\eta * \frac{\partial E}{\partial w_{ij}}\right)$$

weight increment — learning rate — weight gradient

$$\Delta w_{ij} = \left(\eta * \frac{\partial E}{\partial w_{ij}}\right) + \left(\gamma * \Delta w_{ij}^{t-1}\right)$$

momentum factor — weight increment, previous iteration

McCaffrey, J. (2017). Figure: Change of weights in momentum based update. Adapted from "Back-Propagation Update Without and With Momentum" by J. McCaffrey, 2017. Retrieved from https://visualstudiomagazine.com/articles/2017/08/01/neural-network-momentum.aspx.

In our experiments, we typically use the momentum rate of 0.01, but this can vary for different types

of problems.

The below figure shows a visualisation of the gradients of momentum and canonical gradient step. An alternative momentum update known as Nesterov momentum is also shown.
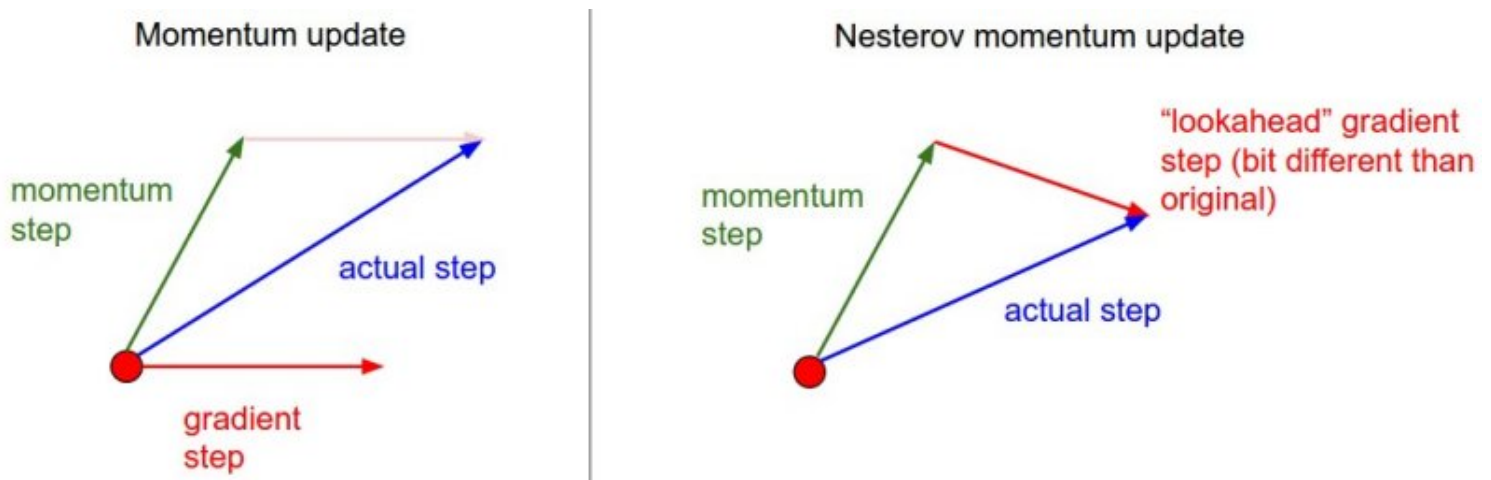

Figure: A visualisation of the gradients of momentum and canonical gradient step. Adapted from "CS231n Convolutional Neural Networks for Visual Recognition" by CS. Stanford, 2020. Retrieved from https://cs231n.github.io/neural-networks-3/.

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

> "Nesterov Momentum is a slightly different version of the momentum update that has recently been gaining popularity. It enjoys stronger theoretical converge guarantees for convex functions and in practice, it also consistently works slightly better than standard momentum." (Stanford CS, 2020)

```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

Github. (2020). CS231n Convolutional Neural Networks for Visual Recognition. Retrieved from https://cs231n.github.io/neural-networks-3/.

> i Watch the below video to understand momentum in more detail.

Momentum detail explanation with examples

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

AIQCAR (2019, May 30). *Momentum Detail Explanation with Examples* [online video]. Retrieved from: https://www.youtube.com/watch?v=6iwvtzXZ4Mo.

The below code shows how the momentum is implemented.

```python
1  def BackwardPass(self, input_vec, desired):
2          out_delta =   (desired - self.out)*(self.out*(1-self.out))
3          hid_delta = out_delta.dot(self.W2.T) * (self.hidout * (1-self.hi
4          #https://www.tutorialspoint.com/numpy/numpy_dot.htm  https://www
5
6          if self.vanilla == True: #no momentum
7              self.W2+= self.hidout.T.dot(out_delta) * self.learn_rate
8              self.B2+=  (-1 * self.learn_rate * out_delta)
9
10             self.W1 += (input_vec.T.dot(hid_delta) * self.learn_rate)
11             self.B1+=  (-1 * self.learn_rate * hid_delta)
12         else: # use momentum
13             v2 = self.W2.copy() #save previous weights http://cs231n.git
14             v1 = self.W1.copy()
```

Stochastic Gradient Descent

Stochastic Gradient Descent

Vanilla Gradient Descent

So far we have been using gradient descent-based weight update that uses the entire training data sets as a single batch. The training data set can be divided into mini-batches and gradient descent can be applied to them as usual. Consider the example below that shows batch gradient descent or vanilla gradient descent that considers the entire training dataset as a batch that we have been using so far. See the pseudocode below.

```python
1  for i in range(number_epochs):
2      grad = evaluate_gradient(loss_function, data, params)
3      weights = weights - (learning_rate * gradients)
4
5  #vanilla gradient descent
```

Stochastic gradient descent

Stochastic Gradient Descent (SGD) computes the gradient for an instance of the dataset and then updates weight after gradients are computed. This is different from vanilla gradient descent where the gradients are first computed for the entire batch and then the weight is updated afterward. Note data at every epoch is shuffled. See the pseudocode below.

```python
1  for i in range(number_epochs):
2      np.random.shuffle(data)
3
4      for example in data:
5          grad = evaluate_gradient(loss_function, data, params)
6          weights = weights - (learning_rate * gradients)
7
8  #stocastic gradient descent
```

Minibatch gradient descent

Mini-batch gradient descent combined vanilla and stochastic gradient descent using small or mini-batches to reduce the variance of the parameter updates for stable convergence. This is very useful for big data and deep learning models especially in terms of memory management and other technical issues since the entire batch (which can be dataset more than 100 000 in size) is not used at once. See the pseudocode below.

```python
1  for i in range(number_epochs):
2      np.random.shuffle(data)
3
4      for each minibatch in get_batches(data, batch_size=50):
5          grad = evaluate_gradient(loss_function, data, params)
6          weights = weights - (learning_rate * gradients)
7
8  #minibatch gradient descent
```

Source: S. Ruder, "An overview of gradient descent optimization algorithms":
https://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent

Further reading:

1. S. Ruder, "An overview of gradient descent optimization algorithms":
   https://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent
2. https://www.geeksforgeeks.org/difference-between-batch-gradient-descent-and-stochastic-gradient-descent/?ref=rp

Example of SGD and Momentum (from scratch)

The SGD is applied with and without data shuffle. Minibatch gradient descent can be implemented in this code; however, you will need to first update the GD training where gradients are calculated for the entire training dataset and then the weight update is done later.

```python
1
2  #useStocastic = True # False for vanilla BP with SGD (no shuffle of data
3
4  #updateStyle = False # True for Vanilla SGD, False for momentum SGD
5
6  def BP_GD(self, trainTolerance):
7          Input = np.zeros((1, self.Top[0])) # Temp hold input
8          Desired = np.zeros((1, self.Top[2]))
9
10         #minibatchsize = int(0.1* self.TrainData.shape[0]) # Choose a mi
11
12         Er = []
13         epoch = 0
14         bestRMSE= 10000 # Assign a large number in begining to maintain
```

Next, we use our code to demonstrate the performance of momentum and stochastic gradient descent for the Iris problem. Note that the complete code is available in the next lesson.

Sample results for the Iris problem

We note that there are implementations where the data shuffle is not explicitly done in SGD. Hence we would like to evaluate what effect shuffle has on the performance of SGD. Here are some results.

SGD (no shuffle) + no momentum

#useStocastic = False (notation in code in the next lesson)

#updateStyle = True

The below figure presents the results for the Iris problem that shows the RMSE (error) over time (epochs on the x-axis).
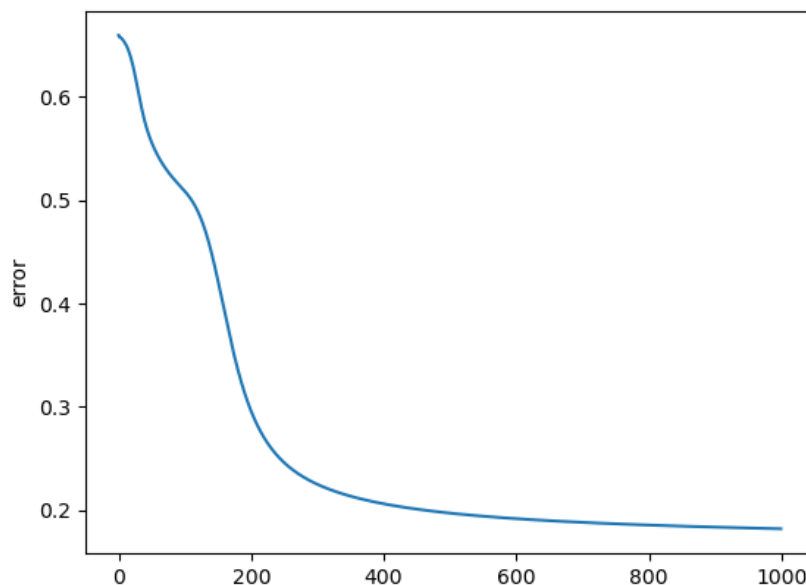

Figure: RMSE error for gradient descent.

Results for 5 experimental runs:

- classification performance in % (mean and std): 93.45 0.68 (train), 95.0 0.0 (test)
- computational time taken (mean and std): 3.33 0.087 (seconds)
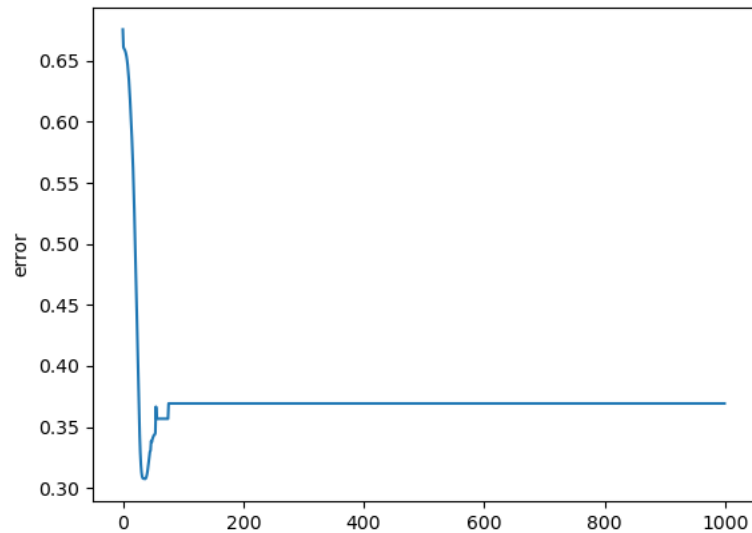
#Good train and test performance but poor time.

SGD (no shuffle) + momentum

#useStocastic = False

#updateStyle = False

The below figure presents the results for the Iris problem that shows the RMSE (error) over time (epochs on the x-axis)

Figure: RMSE error for SGD.

Results for 5 experimental runs:

- classification performance in % (mean and std): 68.0 36.44 (train), 79.0 27.32 (test)
- computational time taken (mean and std): 3.55   1.70 (seconds)

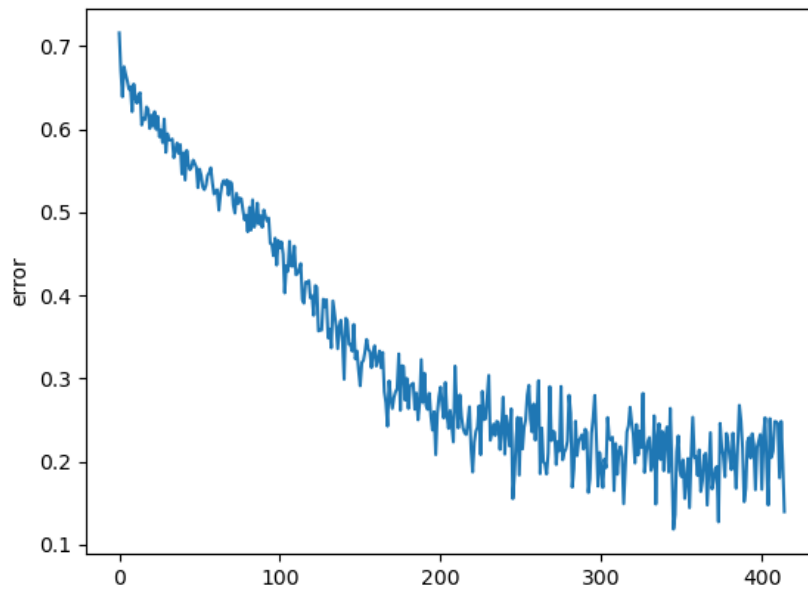#Poor train and test performance with poor time.

SGD (shuffle) + no momentum

#useStocastic = True

#updateStyle = True

The below figure presents the results for the Iris problem that shows the RMSE (error) over time (epochs on the x-axis)

Figure: RMSE (error) for GD and Momentum.

Results for 5 experimental runs:

- classification performance in % (mean and std): 96.0 0.44 (train), 96.5 1.22 (test)
- computational time taken (mean and std): 1.72 0.23 (seconds)

#Good train and test performance in reasonable time.

SGD (shuffle) + momentum

#useStocastic = True

#updateStyle = False

The below figure presents the results for the Iris problem that shows the RMSE (error) over time (epochs on the x-axis).
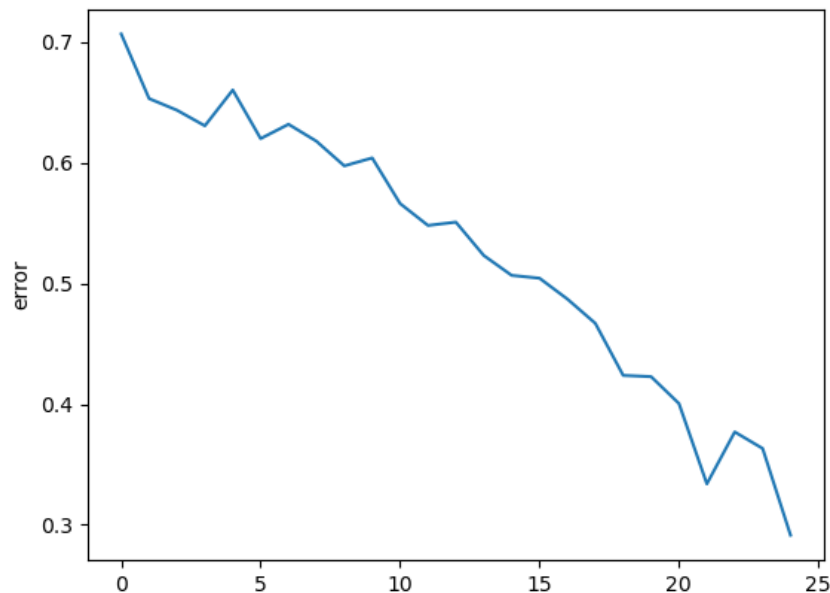
Figure: RMSE (error) for SGD and Momentum.

Results for 5 experimental runs:

- classification performance in % (mean and std): 76.54 38.27 (train), 95.0 2.73 (test)
- computational time taken (mean and std):   1.95 2.21(seconds)

#Chaotic train performance with high variance in results but good generalisation performance.

Discussion of results

The results above show that SGD with shuffle has a chaotic behaviour looking at the error over time when compared with SGD without shuffle. When Momentum is used, both versions of SGD have a much smoother surface in terms of convergence.

This is just an example to show how to conduct an experiment to evaluate the methods. Note that all experiments used the same neural network topology in terms of hidden layers and the same activation function. A better experiment would consider at least 30 experimental runs to reach better conclusions.

Momentum and SGD in Python

The below code demonstrates the use of SGD and Momentum. The SGD is applied with and without data shuffle, you can choose between them by configuring Line 249 and 251 of the below code and also use different momentum rate (Line 252). You can also change the learning rate and the number of experiments. Moreover, you can try for different problems. Also, test for the different number of hidden neurons.

You can make a short report of your experiment and post your results in the discussion forum (Week 4 discussion activity) to compare your results with those of your classmates.

Neural networks with scikit-learn

Now that you have learned the basics of neural networks by covering fundamental issues such as gradient descent, weight update, and training procedure, it is good to know which approaches are being used most commonly in the research sector and industry.

The code that you worked on earlier was an implementation by R. Chandra which has been well-tested in several research projects but it has some limitations such as extending the number of hidden layers. It is important to learn existing libraries so you can test other functionalities. There are several different variations of multilayer perceptron in terms of different gradient-based algorithms and topologies.

Scikit-learn

Scikit-learn is a machine learning library based on Python that features neural networks and associated training algorithms. It has been very popular in the industry, and it's good to learn to apply it to the problems studied previously and also to some new problems.

The below code demonstrates the use of scikit-learn for a multilayer perceptron. Note how the scikit-learn library is utilised (Lines 7-10). Lines 12-18 shows which components of the library are used to import the datasets and evaluation metrics.

Observe how the data is trained and the test split is created (Line 37). The neural network model is created with 3 hidden layers of 8 neurons (Line 43) in each hidden layer with stochastic-gradient descent algorithm and different activation functions.

You can also use other training algorithms such as Adam which will be covered later. Lines 51-55 show how the results are reported with confusion matrix etc., all utilised directly from the library.

Diabetes data set so that you know how to use your own data sets rather than just relying on the built-in or processed data given by scikit-learn.

Further reading:

1. Supervised neural networks

2. Scikit-learn for machine learning

3. Machine learning-Diabetes

Neural networks with Keras and TensorFlow

> ℹ️ In this section, you will understand and apply neural networks using a machine learning platform called TensorFlow and application programming interface (API) Keras.

Keras

> "Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result as fast as possible is key to doing good research."
>
> Keras.io (n.d.) About Keras. Retrieved from: https://keras.io/about/.

> ℹ️ Watch the below video that gives an overview of Keras.

Getting started with Keras

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Google Cloud Platform (2018, August 14). *Getting Started with Keras* [online video]. Retrieved from https://www.youtube.com/watch?v=J6Ok8p463C4.

Let's understand the Keras example (Credit card activity) that explains the Kaggle Credit Card Fraud Detection which is an imbalanced class data set. You can copy the code in a new workspace and try to follow the instructions and run the code linked below.

Run the code

The below video gives details of TensorFlow, which is a building bock of Keras that implements machine learning and neural network libraries.

TensorFlow

> **i** Watch the below video on TensorFlow.

TensorFlow in 10 minutes

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

edureka!(2019, May 6). *TensorFlow In 10 Minutes | TensorFlow Tutorial For Beginners | Deep Learning & TensorFlow | Edureka* [online video]. Retrieved from: https://www.youtube.com/watch?v=tXVNS-V39A0.

Further reading:

1. Tensorflow example
2. Tensorflow with Keras example
3. Keras examples - momentum and learning rate

Keras neural network

We discussed Keras neural network library in the previous lesson. Keras is an alternative to scikit-learn. It focusses more on neural networks and has more features, training algorithms, and neural network architectures. Keras uses TensorFlow an open-source library released by Google to provide effective tensor-based computation and it also uses graphic processing units (GPUs).

The below code shows an example of a Keras neural network that generates its own training data about two different types of circles as shown in the figure below.
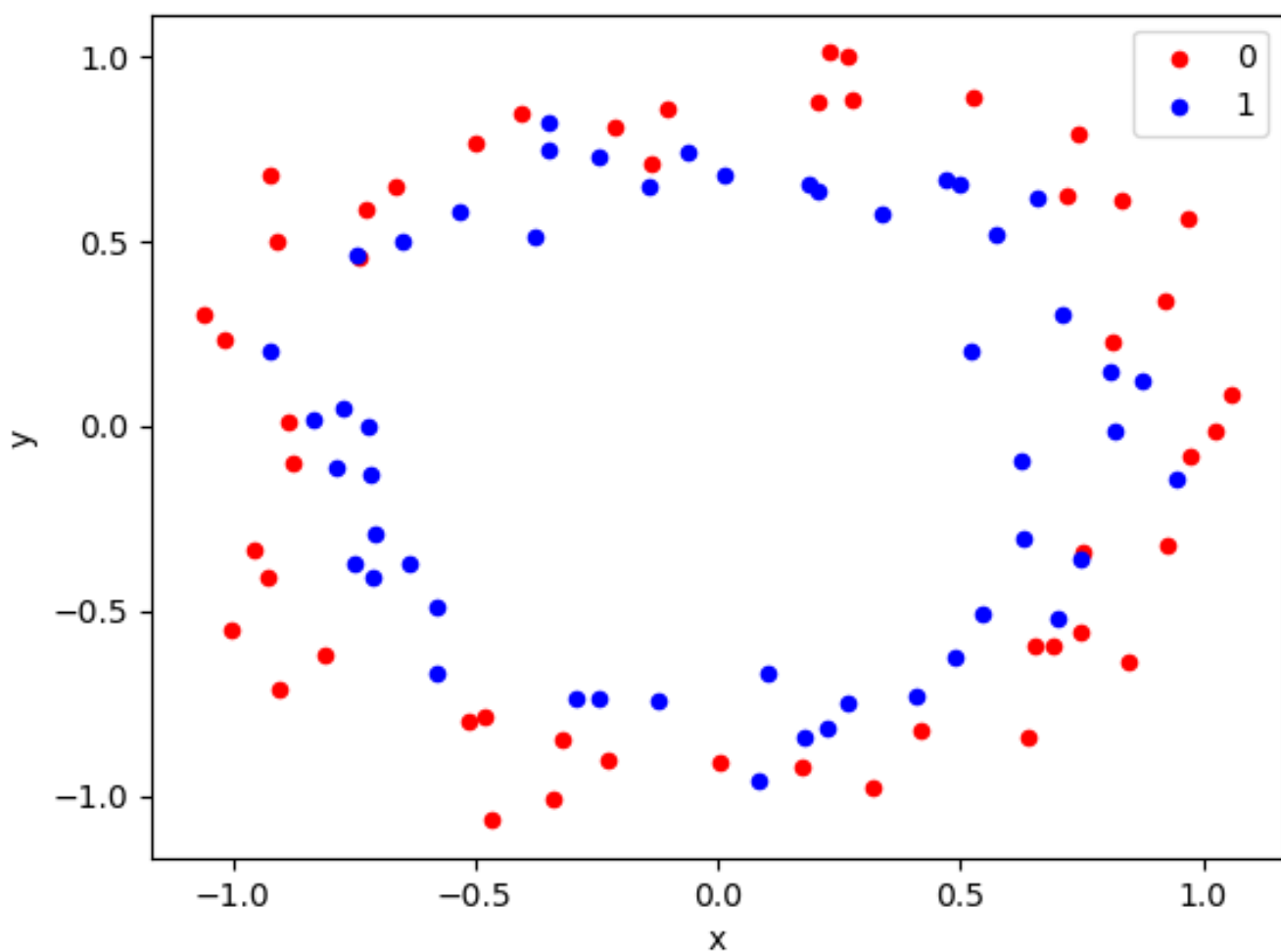

Figure: Data visualisation. Adapted from "How to reduce overfitting with dropout regularization" by J. Brownlee, 2019. Retrieved from https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropout-regularization-in-keras/.

Challenge: Try to use the below code to test other data sets such as Iris, Wine, Diabetes and other data sets of your choice. Note the performance and share it with peers via a discussion thread. Show performance of different optimisers such as SGD and Adam. You can also compare these results with scikit-learn library and report what the similarities and differences are. You can also compare the

computational time of the different libraries as well.

Neural network for time series prediction and regression

So far, we have mostly focused on applications that featured classification problems. Now we look at applications that feature regression problems. Time series prediction can be seen as a type of regression problem (Brownlee, 2017).

Time series prediction employs statistical and machine learning models which include neural networks to predict the future trends of time series. The time series data can be univariate or multivariate. Time series applications or examples include weather prediction, stock market prediction, air pollution prediction, and many others. Can you think of others? We generate time series every day with our activities and devices.

Data processing

Below is an example of a univariate time series, the S&P 500 Stock Market Index.

Figure: S&P Dow Jones Indices LLC. Adapted from "S&P Dow Jones Indices LLC S&P 500 [SP500]" by FRED, Federal Reserve Bank of St. Louis, n.d. Retrieved from https://fred.stlouisfed.org/series/SP500.

We need to process univariate time series data by transforming the values between [0,1] and then apply windowing to transform a 1D time series vector into a NxD vector where D represents the dimension and N represents the number of windows. T represents the time lag in the process of embedding that is based on Taken's theorem.

E.g., consider the following indices for the original time series vector:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

After reconstruction or embedding where D=3 and T=2, we get N=6:

1. **0, 1, 2** 3

2. **2, 3, 4,** 5

3. **4, 5, 6,** 7

4. **6, 7, 8,** 9

5. **8, 9, 10,** 11

6. **10, 11, 12,** 13

We have 3 inputs highlighted in bold and the goal is to predict the 4th value (3, 5, 7, 9, 11, 13).

Can you make the data set if D=3 and T=1? What would be N?

Can you write a Python code to do it?

Prediction with neural networks

Let's see some examples of univariate time series and discuss how neural networks can be used to model them. You can apply the same strategy to multivariate time series as well. The below code explains how a neural network from scikit-learn library is used to model and predict the laser time series. The code will be available to practice in the lessons to follow.

```python
1  import sklearn
2  import numpy as np
3  from sklearn.neural_network import MLPRegressor
4
5  def rmse(pred, actual):
6      return np.sqrt(((pred-actual)**2).mean())
7
8  def main():
9
10     for i in range(1, 4) :
11         problem = i
12         if problem ==    1:
13             traindata = np.loadtxt("Data/Lazer/train.txt")
14             testdata   = np.loadtxt("Data/Lazer/test.txt") #
```

The following examples discuss time series prediction problems from known data sets. The codes will be provided in the next lesson for practice.

Example 1: Sunspot time series features the number of monthly sunspots from 1900 to 1990. Sunspots are temporary phenomena on the Sun's photosphere that appear as spots darker than the surrounding areas caused by concentrations of magnetic field flux that inhibit convection. Sunspots usually appear in pairs of opposite magnetic polarity and their number varies according to the 11-year solar cycle(approx).

Typically, the first half of the data set is used for training and the second half for testing. The below figure shows the sunspot time series (test data set) where the original time series is shown in blue

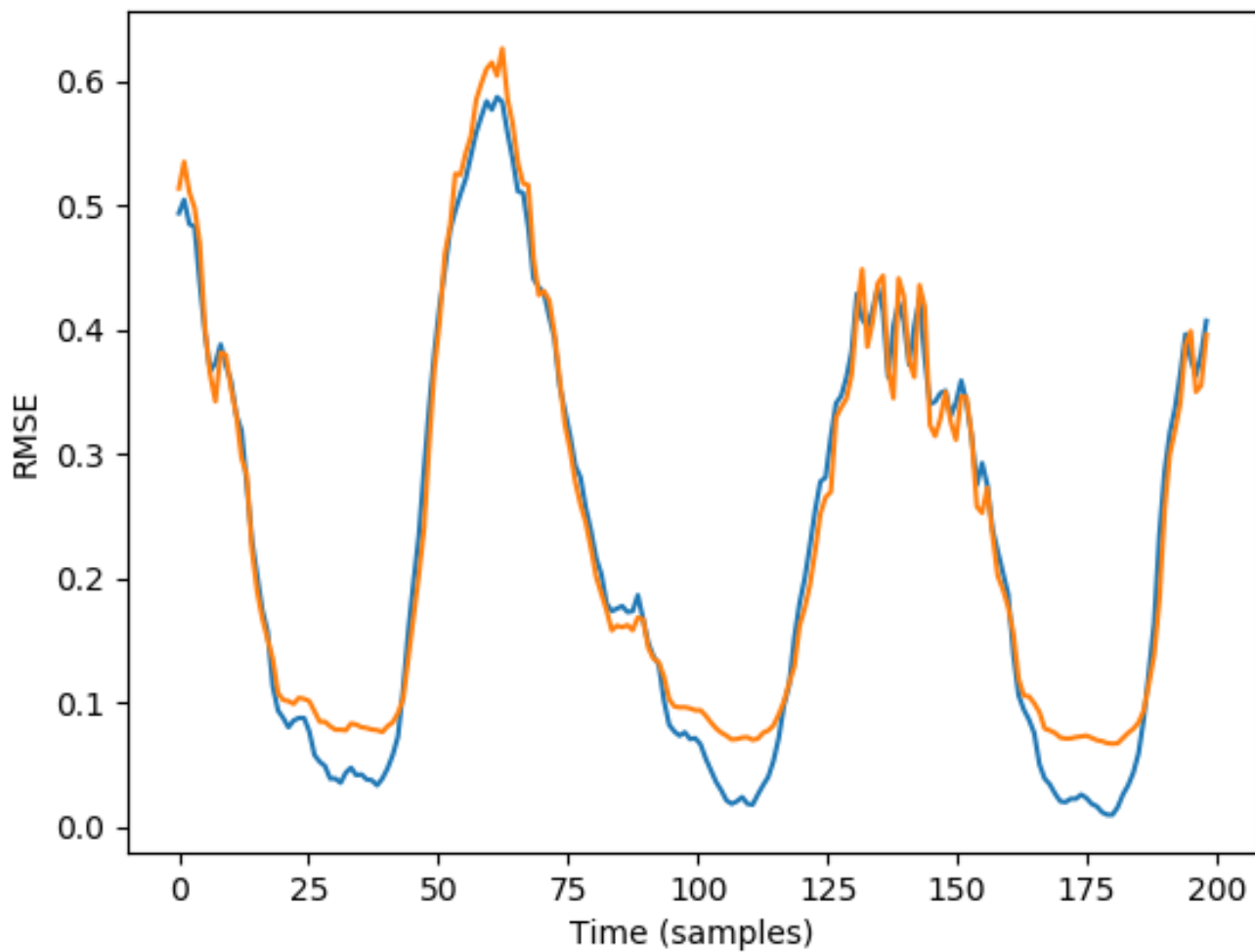and the prediction by the trained neural network is shown in orange.

Figure: Time series analysis for the sunspots. Adapted from "Time series analysis for the sunspots amount from 1900 to 1983" by Github, 2018. Retrieved from https://ionides.github.io/531w18/midterm_project/project9/531mid.html. (Note the original figure has been altered by R. Chandra.)

Example 2: Lazer time series is taken from Santa Fe time series competition.

The figure below shows Lazer time series (test data set) where the original time series is shown in blue and the prediction by the trained neural network is shown in orange.
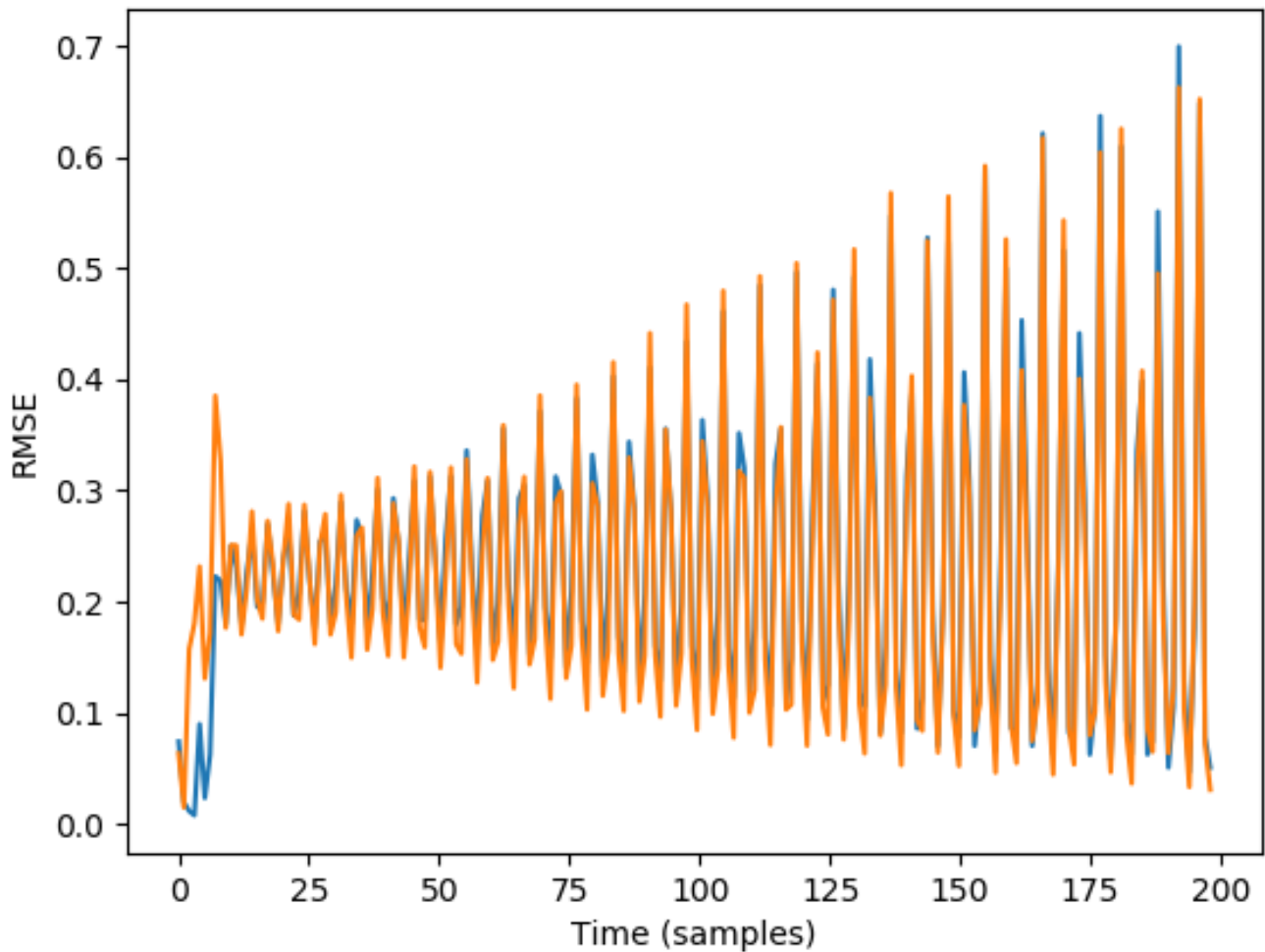
Figure: Lazer time series. Adapted from "SantaFe.A: Time series A of the Santa Fe Time Series Competition" by Rdrr.io, 2019 (https://rdrr.io/cran/TSPred/man/SantaFe.A.html). (Note the original figure has been altered by R. Chandra.)

Example 3: Mackey-Glass time series is generated from Mackey-Glass equations that feature the nonlinear time-delay differential equation.

The below figure shows Mackey-Glass time series (test data set) where the original time series is shown in blue and the prediction by the trained neural network is shown in orange.
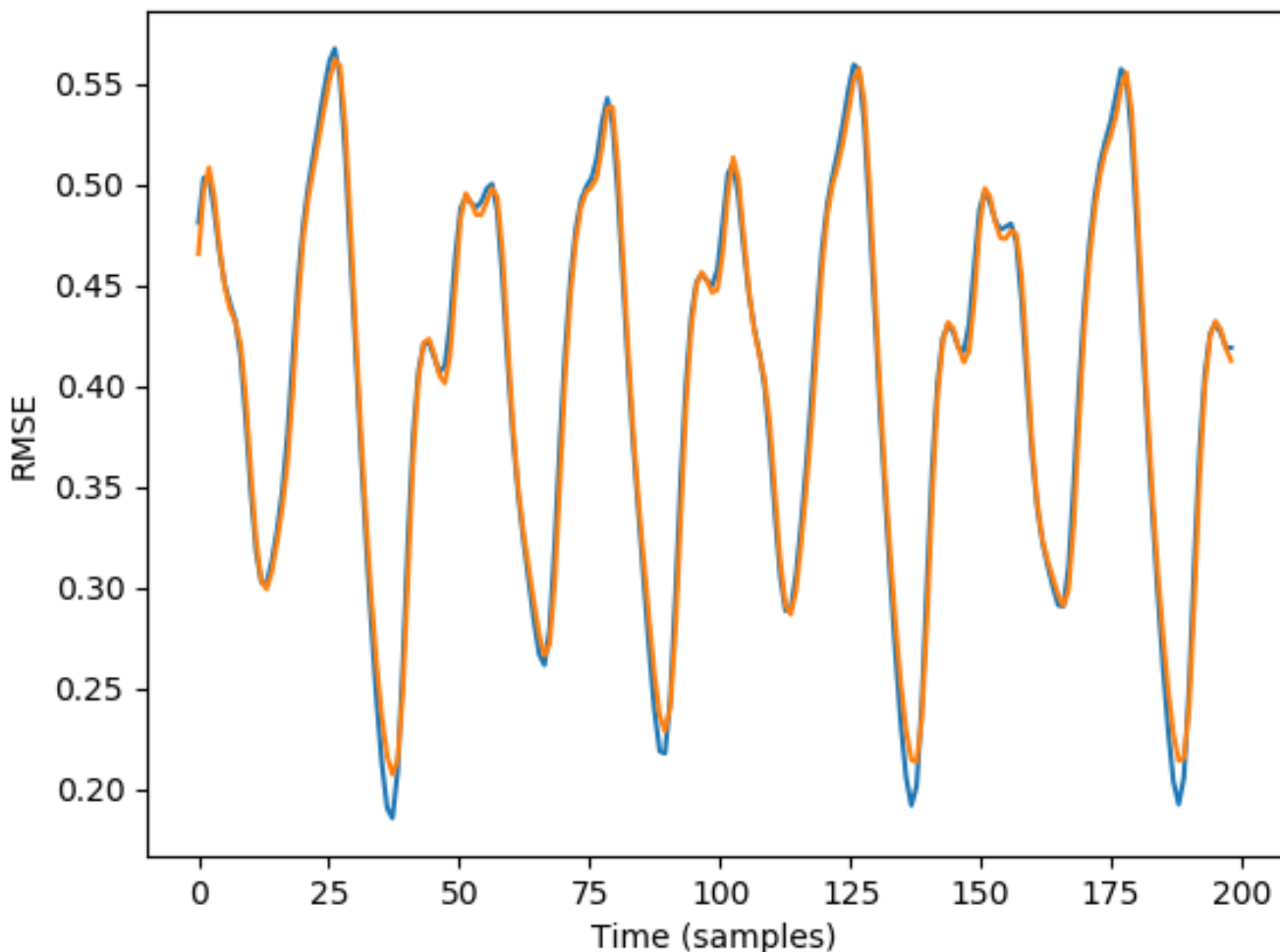
Figure: Mackey-Glass time series, source http://www.scholarpedia.org/article/Mackey-Glass_equation (Note the original figure has been altered by R. Chandra)

OPTIONAL: Further reading

Some research papers on the above time-series data sets:

1. Chandra, R., Jain, K., Deo, R.V., & Cripps, S. (2019). Langevin-gradient parallel tempering for Bayesian neural learning, *Neurocomputing* 359, p 315-329. Retrieved from https://www.sciencedirect.com/science/article/pii/S0925231219308069

2. Chandra, R., Ong, Y.S., & Goh, C.K. (2017). Co-evolutionary multi-task learning with predictive recurrence for multi-step chaotic time series prediction, *Neurocomputing,* 243,p 21-34. Retrieved from https://www.sciencedirect.com/science/article/pii/S0925231217303892

3. Chandra, R. (2015). Competition and Collaboration in Cooperative Coevolution of Elman Recurrent Neural Networks for Time-Series Prediction, *IEEE Transactions on Neural Networks and Learning Systems*, 26(12), p3123-3136. Retrieved from https://ieeexplore.ieee.org/abstract/document/7055352

In the next lesson, we go into details of the above problems with neural network code.

Neural network in scikit-learn for time series prediction

In the previous lesson, we covered different time series problems to train neural networks. This lesson demonstrates those time series problems using scikit-learn to show original data, scaled data set, and reconstructed datasets (train and test) for the different problems.

Let's learn how scikit-learn can be used for the regression problem where we use our time series data. Notice how the data is read and presented to the neural network and how the different training algorithms are used (SGD and Adam). You can use a sigmoid activation function as well.

Note the syntax used:

```
    mlp_sgd = MLPRegressor(hidden_layer_sizes=(5, ), activation='relu', solver='sgd',
 alpha=0.1,max_iter=5000, tol=0)
    mlp_sgd.fit(x_train,y_train)
    y_predicttrain = mlp_sgd.predict(x_train)
    y_predicttest = mlp_sgd.predict(x_test)
    train_acc = rmse( y_predicttrain,y_train)
    test_acc = rmse( y_predicttest, y_test)
```

Random Forest Regression method is discussed for comparison. Random Forest was introduced in Week 2. RMSE is used to measure performance of the model. Note that Adam will be covered in the following lessons.

Chandra, R., Jain, K., Deo, R.V., & Cripps, S. (2019). Langevin-gradient parallel tempering for Bayesian neural learning, *Neurocomputing* 359, p 315-329. Retrieved from
https://www.sciencedirect.com/science/article/pii/S0925231219308069

Neural networks for time series and regression

Let's start with the implementation of a neural network. Note that the implementation of the forward and backward pass remains the same. You used the same code for classification problems in the previous lessons and you will find that some of the variables report the classification performance. This is a regression problem. You can ignore those variables and take note of the RMSE.

The below code helps you to predict the performance. You can see the prediction graphs discussed in the previous lesson(for Sunspot, Lazer and Mackey-Glass).

Challenge: Run the code and show results for different problems and compare the results with scikit-learn implementation given in the previous lesson. You can post your results in a discussion thread to discuss the results with your peers. Note that you should use the same training time. You can also change the number of hidden neurons. You can add more hidden layers in the scikit-learn version and evaluate the prediction performance.

Keras for R

R interface to Keras

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation. It helps to get results with the least possible delay.

Keras has the following key features:

- It allows the same code to run on CPU or on GPU, seamlessly.
- It has a user-friendly API which makes it easy to quickly prototype deep learning models.
- It has built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and a combination of both.
- It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means Keras is appropriate for building any deep learning model from a memory network to a neural Turing machine.
- It is capable of running on top of multiple back-ends including TensorFlow, CNTK, or Theano.

For additional details on why you might consider using Keras for your deep learning projects, read Why Use Keras?.

Keras website provides documentation for the R interface to Keras. Find the relevant information here: https://keras.io.

Please note Ed platform has already integrated Keras with R. To install Keras in RStudio, click on the below link:

https://keras.rstudio.com/

Basic regression tutorial using Keras in R

In a regression problem, we aim to predict the output of a continuous value such as a price or a probability. It is different from a classification problem where we aim to predict a discrete label such as predicting whether a picture contains an apple or an orange.

This notebook builds a model to predict the median price of homes in a Boston suburb during the mid-1970s. To do this, we'll provide the model with some data points about the suburb such as the crime rate and the local property tax rate. Regression problems are similar to time series prediction problems. Click on the below link to understand how to apply regression using Keras in R.

https://keras.rstudio.com/articles/tutorial_basic_regression.html

You can copy and paste the code in your workspace and see how it runs. Note that your assignment (Assessment 3) will focus on applying the existing code for given problems. You can use Python or R

for your assessment.

 The below code discusses a regression problem (Boston housing data set) using Keras neural networks in R.

```r
1  #Source: https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html
2  #Source: https://keras.rstudio.com/articles/tutorial_basic_regression.ht
3
4  library(keras)
5  boston_housing <- dataset_boston_housing()
6  c(train_data, train_labels) %<-% boston_housing$train
7  c(test_data, test_labels) %<-% boston_housing$test
8  paste0("Training entries: ", length(train_data), ", labels: ", length(tr
9
10 train_data[1, ]
11
12 library(tibble)
13
14 column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
```

Below is another example, Keras neural network for a synthetic classification problem.

```r
1
2  #source https://www.datatechnotes.com/2018/08/simple-usage-of-keras-in-r
3
4  library(keras)
5
6  set.seed(12)
7  n=2000   # number of sample data
8  a <- sample(1:20, n, replace = T)
9  b <- sample(1:50, n, replace = T)
10 c <- sample(1:100, n, replace = T)
11 flag <- ifelse(a > 15 & b > 30 & c > 60, "red",
12                 ifelse(a<=9 & b<25& c<=35, "yellow", "green"))
13 df <- data.frame(a = a,
14                  b = b,
```

Advanced gradient methods: Adagrad and Adam

In the first week, you learned about gradient descent. This section introduces you to other gradient descent methods that are adaptive, not constant.

Adaptive learning rate

Note that we used a user-defined constant learning rate both in GD and SGD in all the previous examples. We also demonstrated how simple heuristics (momentum) can be used for some problems. Researchers are trying to find more effective ways to determine the learning rate.

Adaptive gradient is a concept that is based on adapting the learning rate during training.

The below figure shows the effect on the loss function given by different values of the learning rate.
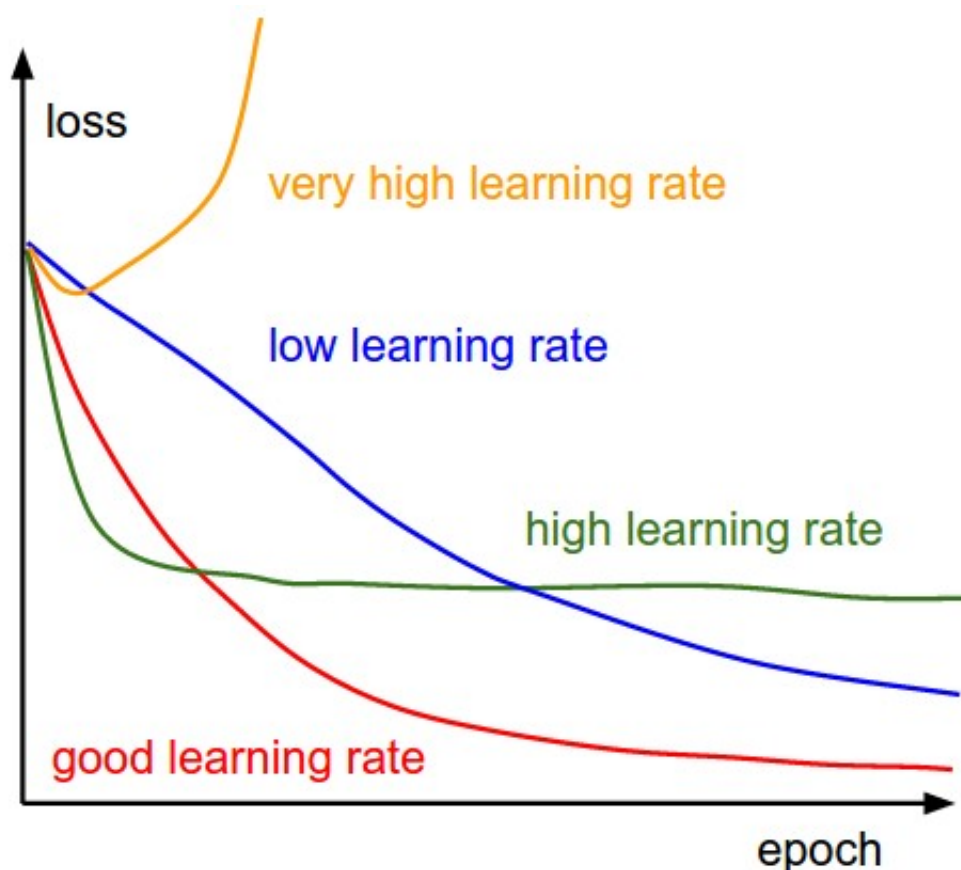

Figure: Learning rates. Adapted from "CS231n Convolutional Neural Networks for Visual Recognition" by CS. Stanford, 2020. Retrieved from https://cs231n.github.io/neural-networks-3/.

Consider a different learning rate for each weight. The high-dimensional non-convex nature of neural networks training could lead to different sensitivity on each input feature that affects the weight update. An optimal learning rate could be too small in some dimension and could be too large in another dimension of the weight update. If we have thousands or millions of weights, then having separate learning rate for each weight would add to the problem, making it far too complex.

Note the rest of the lesson is adapted from: https://ruder.io/optimizing-gradient-descent/index.html

Adaptive gradient algorithm (AdaGrad) algorithm (Duchi, Hzan &Singer, 2011, p 2121) adaptively scales the learning rate for each dimension.

Consider that g_t denotes the gradient at time step t, and $g_{t,i}$ is the partial derivative of the error function with respect to the weights θ_i at time step t

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$$

where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagonal element is the sum of the squares of the gradients from the beginning time, $t = 0$. η is a constant step size which determines the size of the step at each update and most implementations use a default value of 0.01

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Ada-delta gradient algorithm

Based on Adagrad, Adadelta restricts the window of accumulated past gradients to a fixed size, rather than accumulating all of them from the past. In Adagrad, the squared gradient is stored over the past which requires lots of memory. In Ada-delta, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The accumulated average $E[g^2]_t$ at time step t depends as a fraction γ which is similar to the momentum term which is based only on the previous average and the current gradient.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2$$

γ typically is 0.9, furthermore

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_{t,i}$$

ϵ is a very small valute to avoid numberial instability (division by zero)

The above is further updated to eliminate the need to set up a default learning rate. Further infor about this is here: https://ruder.io/optimizing-gradient-descent/index.html

RMSprop

RMSprop and Adadelta have both been developed independently to improve Adagrad's radically diminishing learning rates. RMSprop is identical to the update of Adadelta as shown previously.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_{t,i}$$

Hinton suggests γ of 0.9 learning rate is 0.001

Source:

Lecture 6e RMSprop: Divide the gradient by a running average of its recent magnitude

Adam (adaptive moment estimation) learning algorithm (Cornell University, 2017) differs from classical gradient descent. This algorithm maintains the learning rate for the weight of each network and separately adapted as learning unfolds.

Adam computes individual adaptive learning rates for different parameters from the estimates of first and second moments of the gradients. Adam features the strengths of root mean square propagation (Hinton, n.d.) and AdaGrad.

We note that Adadelta and RMSprop stores an exponentially decaying average of past squared gradients v_t. Adam also keeps an exponentially decaying average of past gradients m_t which is similar to momentum.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$$

where m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients which gives the name of the method. β_1 and β_2 are two hyperparameters (decay rates), β_1 controls first-order momentum and β_2 controls second-order momentum.

Given that m_t and v_t are initialized as vectors of 0's, they would be biased towards zero in the initial time steps, especially when the decay rates are close to 1. Hence, the following update is done:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the Adam update equation can be expressed as follows

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

ϵ is a small scalar used to prevent division by 0.

Here is a summary of the different gradient methods (optimizers):

Optimizers

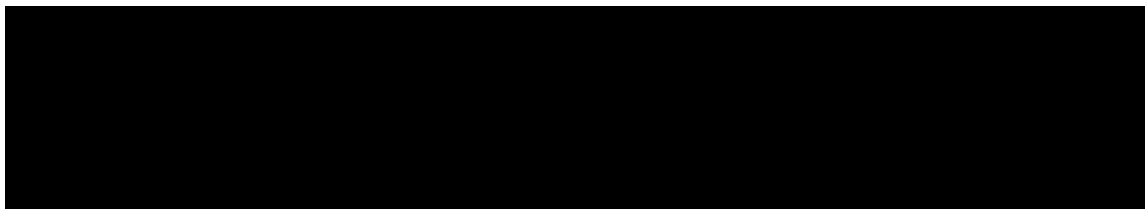|	
Gradient Descent:	$\theta = \theta - \alpha . \nabla_\theta J(\theta)$
Stochastic Gradient Descent	$\theta = \theta - \alpha . \nabla_\theta J(\theta; sample)$
Mini-Batch Gradient Descent	$\theta = \theta - \alpha . \nabla_\theta J(\theta; N\ samples)$
SGD + Momentum	$v = \gamma.v + \eta. \nabla_\theta J(\theta)$ $\theta = \theta - \alpha v$
SGD + Momentum + Acceleration	$v = \gamma.v + \eta. \nabla_\theta J(\theta - \gamma.v)$ $\theta = \theta - \alpha v$
Adagrad	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}} + \epsilon} \nabla_{\theta_{t,i}} J(\theta_{t,i})$
Adadelta	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}]} + \epsilon} \nabla_{\theta_{t,i}} J(\theta_{t,i})$
Adam	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[G_{t,ii}]} + \epsilon} \times E[g_{t,i}]$

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Source of Figure and Video: "Optimizers - EXPLAINED!" https://www.youtube.com/watch?v=mdKjMPmcWjY&feature=emb_logo

1. Adam: A Method for Stochastic Optimisation
2. Lecture 6e RMSprop: Divide the gradient by a running average of its recent magnitude
3. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization
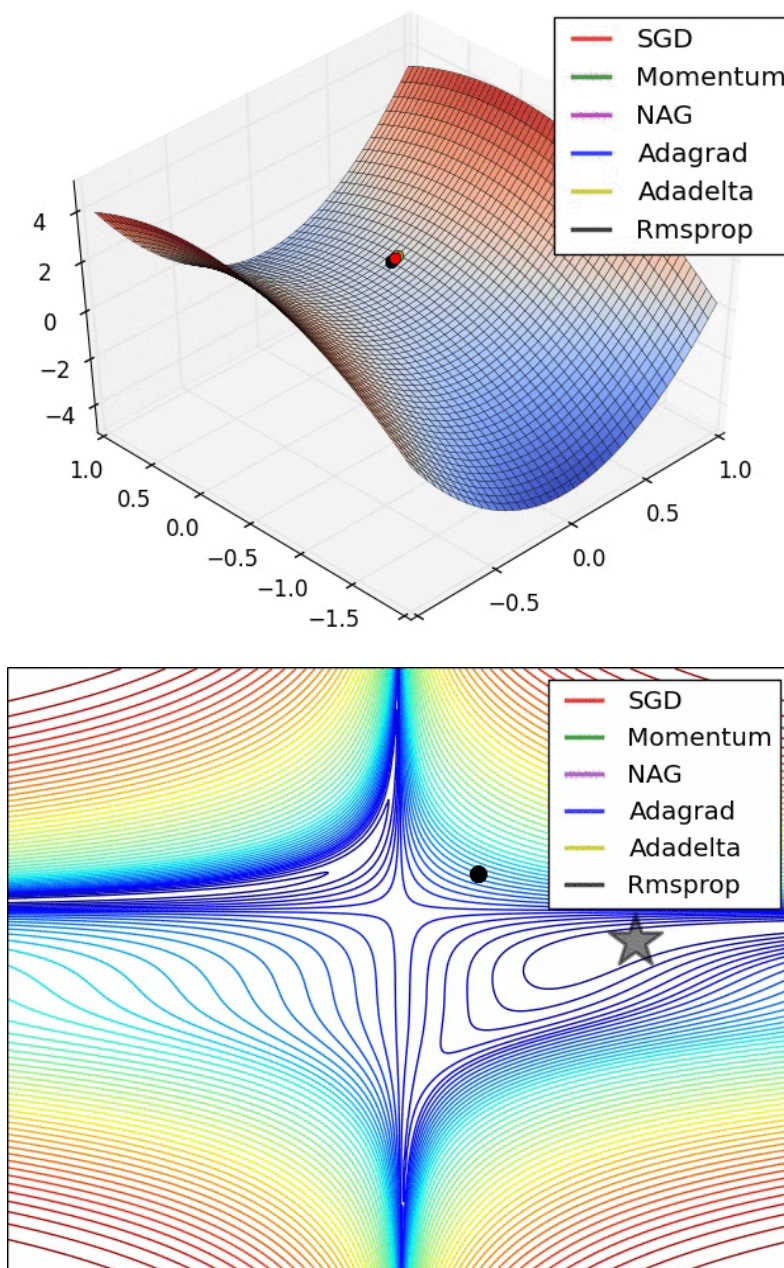4. An overview of gradient descent optimization algorithms

Figure: Performance of different methods. Adapted from "Optimising gradient descent" by S. Ruder, 2016. Retrieved from https://ruder.io/optimizing-gradient-descent/.

L26/1 Momentum, AdaGrad and RMSprop Adam

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Smola, A. (2019, May 7). *L26/1 Momentum, AdaGrad and RMSprop Adam* [online video]. Retrieved from https://www.youtube.com/watch?v=gmwxUy7NYpA.

Below are the few examples of how you can use the different optimisers in your backpropagation neural network. Click on the links.

1. Adagrad
2. RMSprop
3. Adam

Below is an example visualizes some training loss curves for different stochastic learning strategies, including SGD and Adam. Note L-BFGS is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) works for small datasets and not part of this course but good to know about it for information purpose.
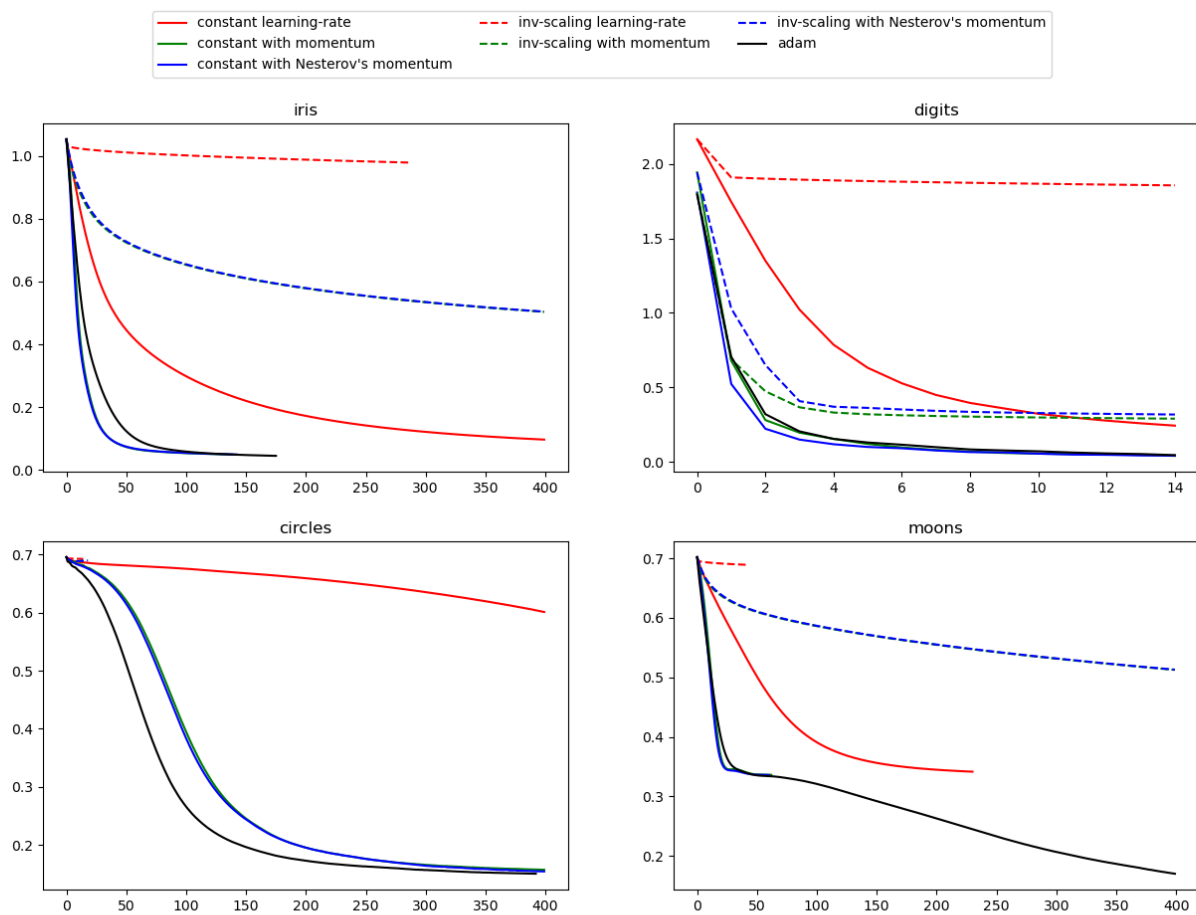
Figure Source. Compare Stochastic learning strategies for MLPClassifier: https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_training_curves.html

Further reading:

1. https://towardsdatascience.com/learning-parameters-part-5-65a2f3583f7d

2. https://keras.io/api/optimizers/

3. https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Exercise 1

Python challenge:

- Try implementing the GD version in NN code from scratch and see the effect of momentum and compare with SGD.
- Extend the neural network from scratch in python code to multi-step time series prediction. Use stocks from the NASDAQ stock market. Use Keras or scikit-learn to do the same and compare your results using Adam and SGD.
- Discuss or try to implement Adagrad/Ada-delta/Adam from scratch for logistic regression/neural networks.

R challenge

- Use Keras in R or any other library and compare the effect of momentum on SGD vs GD.
- Use Keras in R or any other library for multi-step time series prediction. Use stocks from the NASDAQ stock market and compare results using Adam and SGD.

Weight Decay - regularisation

Recall the idea behind L1 and L2 regularisation in logistic regression. You can extend them for neural networks, also known as weight decay which ensures that your trained neural network does not overfit with too large values in the weights. The initial paper that showed the effectiveness of this approach is given here:

Krogh, A., & Hertz, J. A. (1992). A simple weight decay can improve generalization. In *Advances in neural information processing systems* (pp. 950-957): https://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.pdf
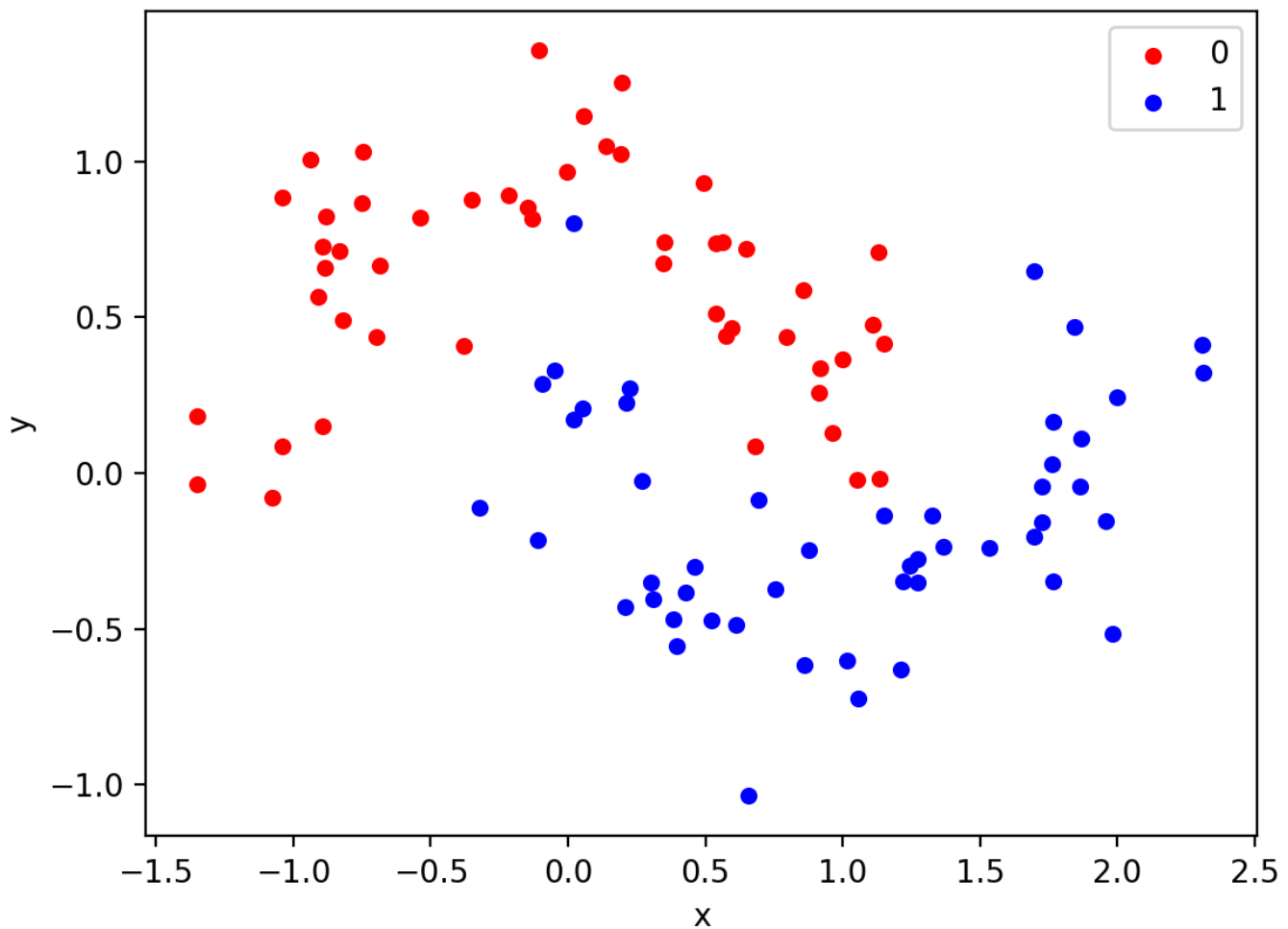
Given original/conventional weight update

$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}$ where η is the learning rate.

Simply, weight decay is given by the following

$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i} - \eta \lambda w_i$ where λ is weight decay constant user defined.

Below is an example using a moon dataset with no regularisation. Data and Figure source: https://machinelearningmastery.com/how-to-reduce-overfitting-in-deep-learning-with-weight-regularization/

```
1 #Source: https://machinelearningmastery.com/how-to-reduce-overfitting-in
2 # overfit mlp for the moons dataset plotting history
3 from sklearn.datasets import make_moons
4 from keras.layers import Dense
5 from keras.models import Sequential
6 from matplotlib import pyplot
7 # generate 2d classification dataset
8 X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
9 # split into train and test
10 n_train = 30
11 trainX, testX = X[:n_train, :], X[n_train:, :]
12 trainy, testy = y[:n_train], y[n_train:]
13 # define model
14 model = Sequential()
```

Below is an example with weight decay based on L2 regularisation:

```python
1  #source: https://machinelearningmastery.com/how-to-reduce-overfitting-in
2  # mlp with weight regularization for the moons dataset plotting history
3  from sklearn.datasets import make_moons
4  from keras.layers import Dense
5  from keras.models import Sequential
6  from keras.regularizers import l2
7  from matplotlib import pyplot
8  # generate 2d classification dataset
9  X, y = make_moons(n_samples=100, noise=0.2, random_state=1)
10 # split into train and test
11 n_train = 30
12 trainX, testX = X[:n_train, :], X[n_train:, :]
13 trainy, testy = y[:n_train], y[n_train:]
14 # define model
```

Now you can see a video that summaries weight decay using L1 and L2 regularisation, apart from dropouts.

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

Dropouts for improved generalisation

The need for better generation performance has been a major challenge in neural networks. Research has shown that a single model can be used to simulate a large number of different network architectures by randomly dropping out the nodes during training. This is a regularisation technique known as dropout. It offers a very computationally cheap way to reduce overfitting and improve performance. Initially, the dropout method was introduced for deep neural networks but this method can also be used for simple neural networks or multilayer perceptions. The below figure shows a neural network before and after dropout during training.

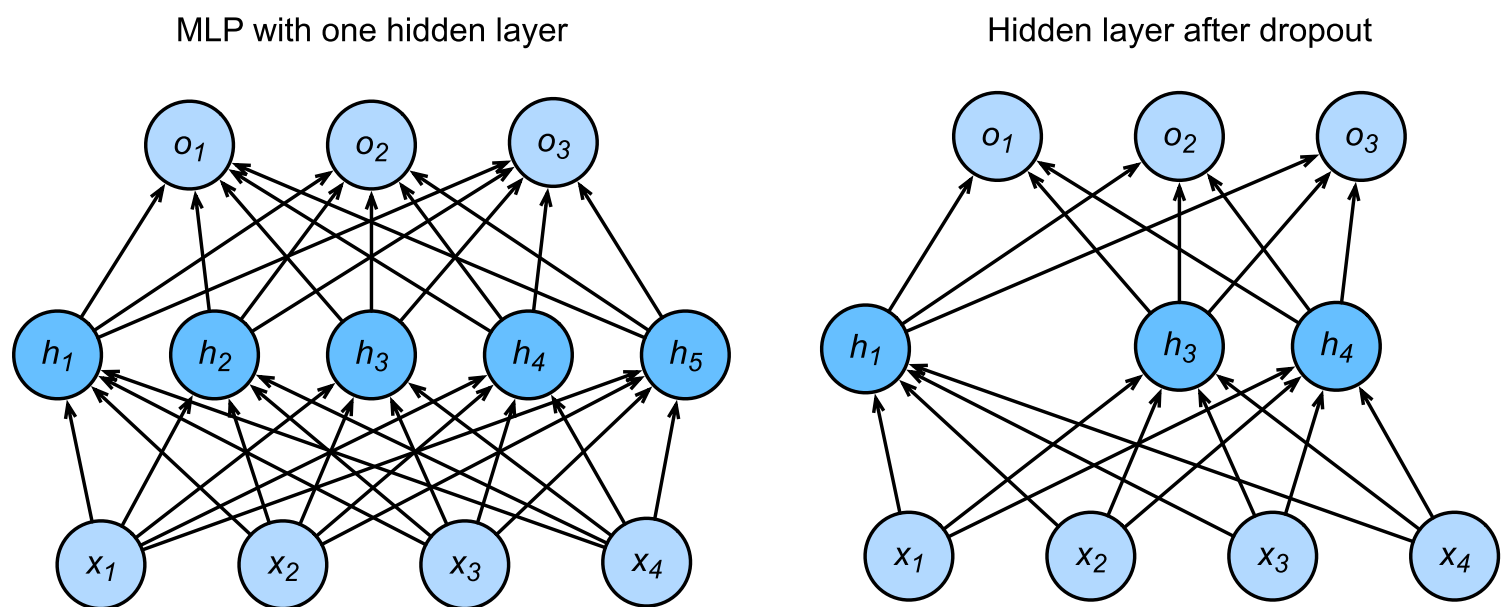Figure: A neural network before and after dropout during training. Adapted from "Dive into Deep Learning" by M.Li, Z. Lipton, A. Smola, & A. Zhang, 2020. Retrieved from https://d2l.ai/chapter_multilayer-perceptrons/dropout.html.

Additional information: Jimmy, B., and Brendan, F. (2013). Adaptive dropout for training deep neural networks. *Advances in neural information processing systems*. Retrieved from http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?.

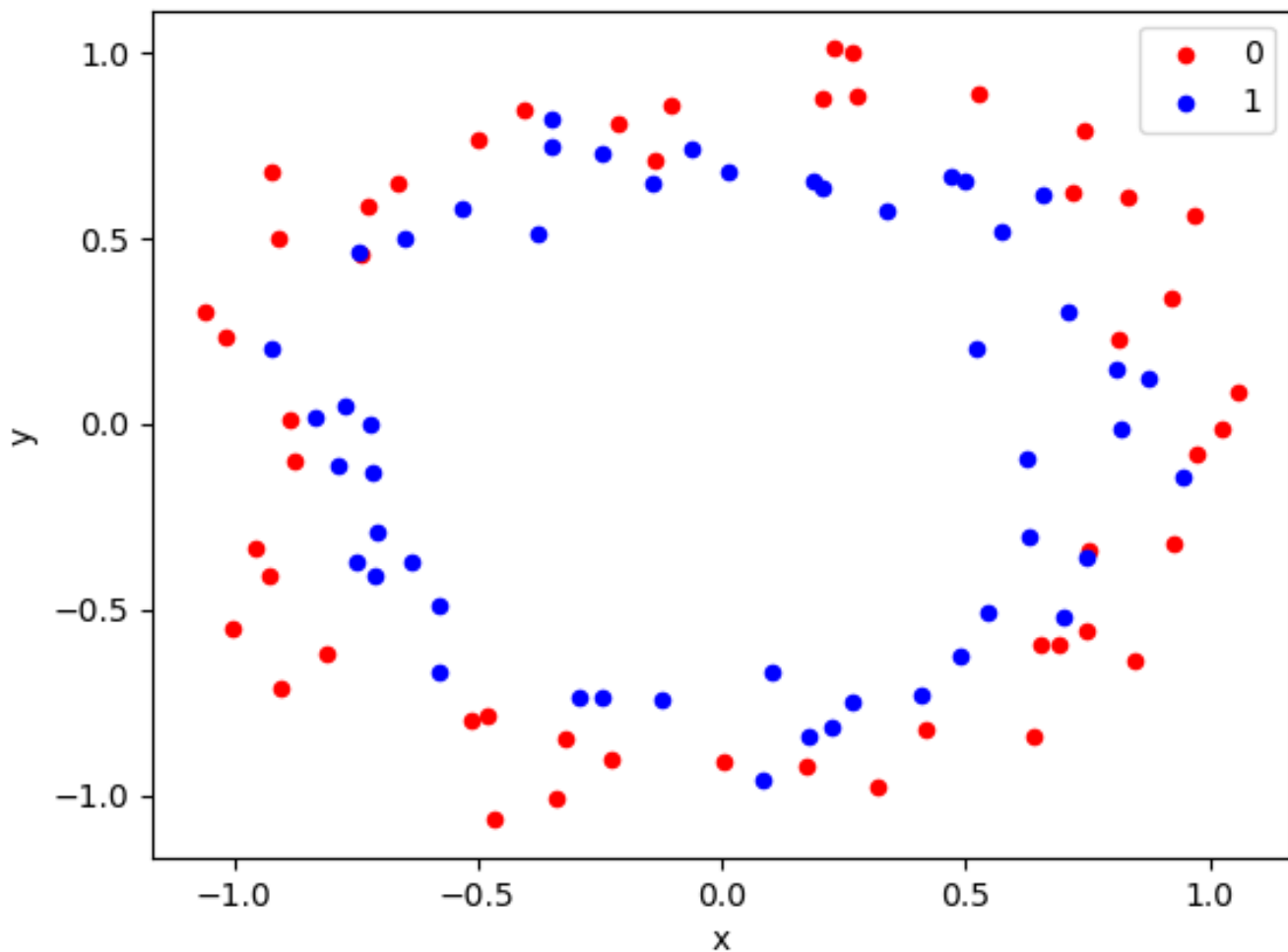The below figure shows a synthetic data set with two different classes of circles.

Figure: training data. Retrieved from https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropout-regularization-in-keras/.

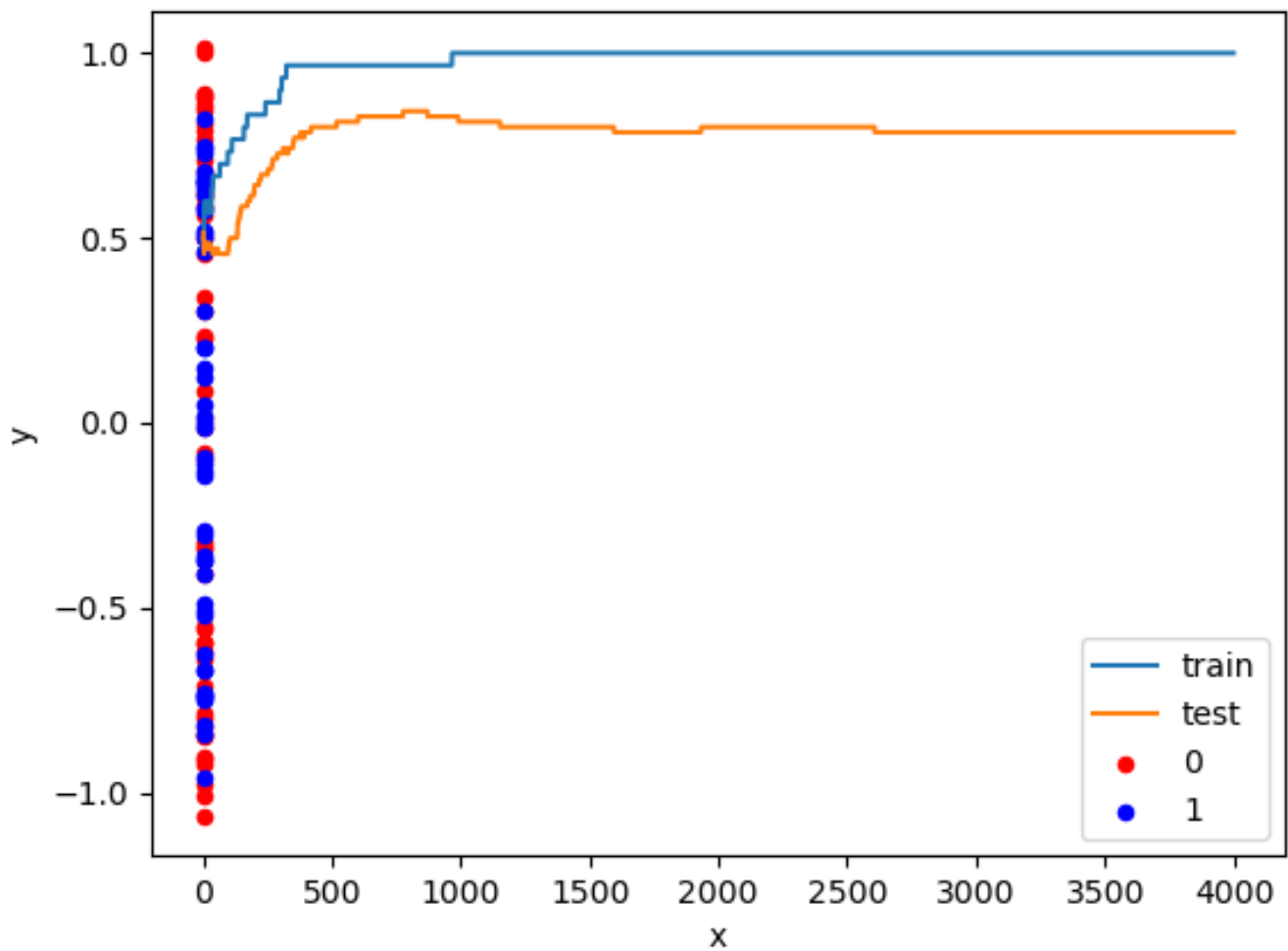The below code shows the training of a neural network without dropout.

```python
1
2 #https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropo
3
4 # no dropout
5 from sklearn.datasets import make_circles
6 from keras.layers import Dense
7 from keras.models import Sequential
8 from matplotlib import pyplot
9
10 from pandas import DataFrame
11
12 # generate 2d classification dataset
13 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
14 # split into train and test
```

Results: The below figure shows the training and test performance over time with a maximum of 4000 epochs.

Now we use the same circle data set to train a neural network with the dropout method.

```
1
2  #https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropo
3
4
5  # mlp with dropout on the two circles dataset
6  from sklearn.datasets import make_circles
7  from keras.models import Sequential
8  from keras.layers import Dense
9  from keras.layers import Dropout
10 from matplotlib import pyplot
11 # generate 2d classification dataset
12 X, y = make_circles(n_samples=100, noise=0.1, random_state=1)
13 # split into train and test
14 n_train = 30
```
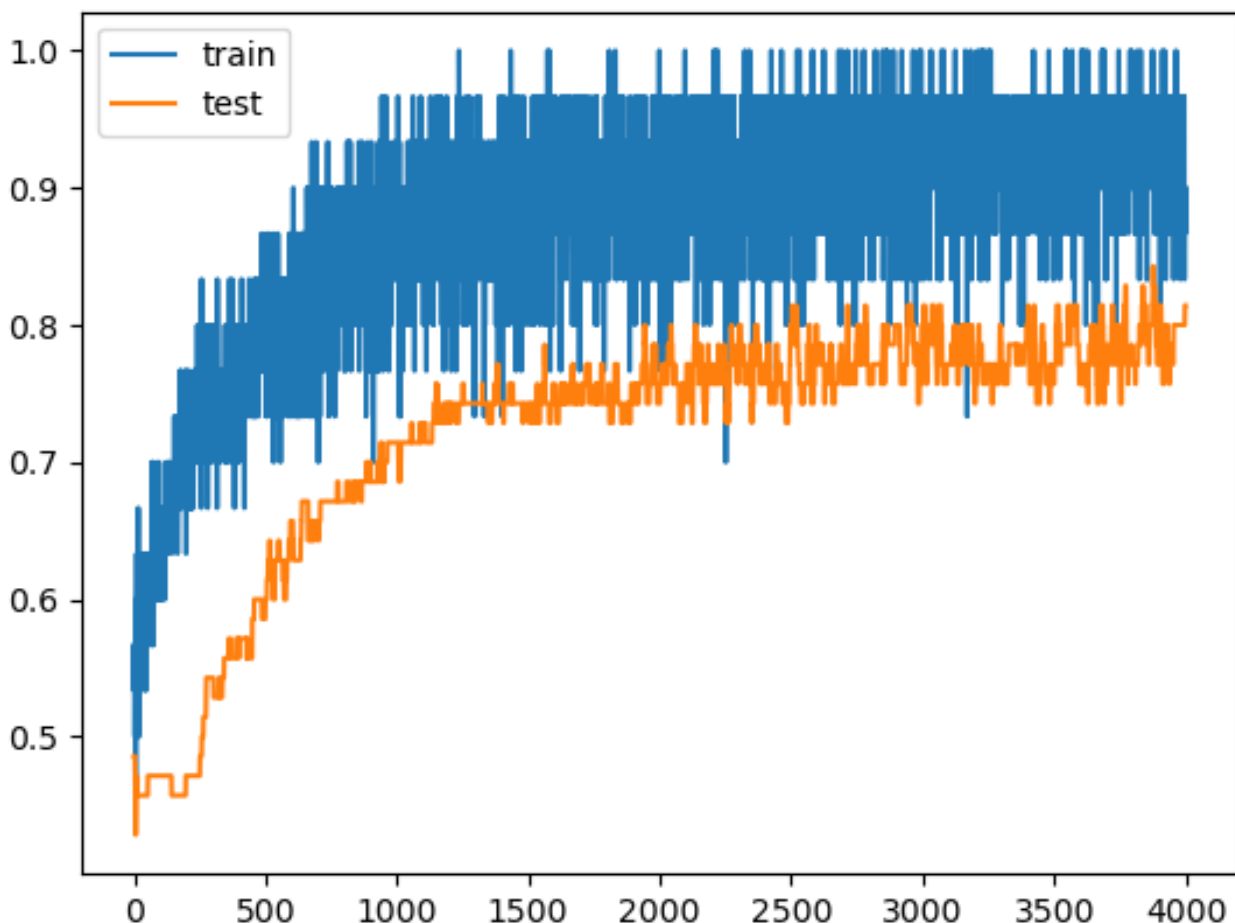

The above results show that the dropout based training archives better generalisation performance over time. Notice how dropout training is rather chaotic when compared to the non-dropout graph shown earlier. This implies that the neural network traverses a wider search space in this case.

Challenge: Can you use the above code with other data sets (such as Iris, Diabetes or Wine data sets) and evaluate the impact of the dropout method?

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

K. Naik. (2019, July 24). *Dropout and Regularisation* [online video]. Retrieved from https://www.youtube.com/watch?v=XmLYl17DbbA.

Exercise 2

Use either R or Python, with Keras and see the effect of Adam vs SGD for any Classification and Regression problem selected from UCI ML repository.

1. Then, apply dropouts and compare the generalisation performance.
2. Compare the performance of dropouts with weight decay.