**Introduction**

This article assumes some knowledge on machine learning and data preparation including k-nearest-neighbors, clustering, neural networks, linear regression, scaling, encoding, imputation, feature engineering, and some math.

Many cases involve *periodic* (or *cyclical*) features. These are features that naturally have looping values; for example, time of day (24-hour cycle), seasons, angles of rotating bodies, virtual wraparound spaces, and most anything sinusoidal. Acceptable methods for handling these values are dependent on the algorithms employed. It is not appropriate to treat these attributes the same way as normal one-dimensional attributes because that makes the minimum and maximum values far apart spacially\*, when they should be close.

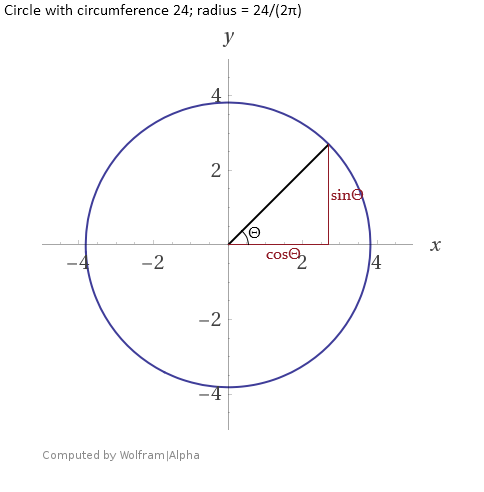
\*The formal spelling is “spatial”, but “spacial” is sometimes chosen because it more clearly references “space”.

**Neural Networks and Linear Regression**

The most common solution to use periodic data in a neural network is to transform each periodic attribute to a two-dimensional circle, so that the inputs are the sine and cosine of the angle.

Example for time of day

|  |  |  |
| --- | --- | --- |
| Time | Input A, sinΘ\* | Input B, cosΘ\* |
| Midnight | sin(0) = 0 | cos(0) = 1 |
| 3 AM | sin(π/4) = 0.7071 | cos(π/4) = 0.7071 |
| 9 AM | sin(3π/4) = 0.7071 | cos(3π/4) = -0.7071 |
| 2 PM | sin(7π/6) = -0.5 | cos(7π/6) = -0.866 |
| 8 PM | sin(5π/3) = -0.866 | cos(5π/3) = 0.5 |

\*Multiplying by the radius, 3.8197, would be acceptable but unnecessary for most tasks.

Technically, this distorts the distance between values: Points on opposite ends of the cycle should be π•radius apart but are treated as if they are only 2•radius apart. This is acceptable for a neural network however, because it can find such non-linear relationships. Other supervised learning methods like linear regression also benefit from this approach; in these cases, it may be helpful to keep this distortion in mind while searching for a good model. Modeling sinusoidal relationships with (non)linear regression can be difficult but having some contextual knowledge of the data makes the process much easier. Most all principles and general approaches used in this section apply to linear regression.

Adjusting the radii of the attributes is unnecessary because the final coefficients or weights will fit themselves accordingly. Getting the radii right will usually take more effort than the time doing so saves.

As part of the universal approximation theorem, any neural network with at least one hidden layer, an unspecified number of neurons, and a non-linear activation function can approximate sinusoidal functions. Non-linear activation functions include RELU, which is mostly linear.

Though not common in modern practice, some neural networks have been built with sinusoidal activation functions, like sine and sine cardinal. In theory, there are cases where they would be preferred. <https://www.researchgate.net/publication/3835580_Neural_networks_with_periodic_and_monotonic_activation_functions_a_comparative_study_in_classification_problems>

It is difficult to find information on neural networks predicting periodic values, so I have written a simple network that demonstrates this capability. Other architectures will be better for various tasks; but after testing, this was the best model I found for this fake data. By converting the target angles to their sine and cosine I get much better results. I also tried tanh for the final activation function because it ranges from -1 to 1 but found that a linear activation performs better.

It is easy to convert predicted rectangular coordinates to the single periodic value, named `real\_y\_pred` in the script.

**Fake Data**

import numpy as np

from matplotlib import pyplot as plt

# Make up data

y = np.arange(0, 30\*np.pi, np.pi/7.1)

x = np.column\_stack((np.sin(y), np.sin(y + np.pi/4)))

noise = np.random.normal(0, np.pi/30, len(y))

y = y+noise

y = np.mod(y, 2\*np.pi)

# What data looks like

fig = plt.figure(figsize=(6,5))

ax = fig.add\_subplot(111, xlabel = 'y', ylabel = 'x1 in red, x2 in blue')

ax.scatter(y, x[:,0], color=[1, 0, 0, 0.3])

ax.scatter(y, x[:,1], color=[0, 0, 1, 0.3])

A screenshot of a cell phone

Description automatically generated

**Multilayer-Perceptron**

# Python script

from keras.layers import Input, Dense

from keras.models import Model

from keras.callbacks import EarlyStopping

from sklearn.model\_selection import train\_test\_split

# Train a model

y\_transformed = np.column\_stack((np.cos(y), np.sin(y)))

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,

y\_transformed,

test\_size=0.2)

inputs = Input(shape=(2,))

hidden1 = Dense(8, activation='elu')(inputs)

hidden2 = Dense(4, activation='elu')(hidden1)

outputs = Dense(2, activation='linear')(hidden2)

model = Model(inputs=inputs, outputs=outputs)

model.compile(optimizer='adam',

loss='mean\_absolute\_error',

metrics=['mean\_squared\_error'])

model.fit(x\_train, y\_train,

validation\_data=(x\_test, y\_test),

epochs = 2000,

callbacks = [EarlyStopping(monitor='val\_loss',

mode='min',

patience = 150)])

# Test

y\_pred = model.predict(x\_test)

print("MLP results:")

print("MAE for cos(y) and sin(y) =", np.mean(np.abs(y\_test - y\_pred)))

real\_y\_test = np.mod(np.arctan2(y\_test[:,1], y\_test[:,0]), 2\*np.pi)

real\_y\_pred = np.mod(np.arctan2(y\_pred[:,1], y\_pred[:,0]), 2\*np.pi)

print("MAE for y =", np.mean(np.abs(real\_y\_test - real\_y\_pred)))

# R script

library(tensorflow)

library(keras)

library(tidyverse)

# Fake data

y <- seq(0, 30\*pi, pi/7.1)

df <- data.frame(x1 = sin(y),

x2 = sin(y + (pi/4)),

y = (y + rnorm(length(y), 0, pi/30)) %% (2\*pi))

# What it looks like

df %>%

ggplot(aes(x = y, y = x1)) +

geom\_point(color = 'red', alpha = 0.5) +

geom\_point(aes(y = x2), color = 'blue', alpha = 0.5) +

labs(y = 'x1 in red, x2 in blue')

# Add trigonometric transformation to target value

df <- df %>%

mutate(cos\_y = cos(y),

sin\_y = sin(y))

# Train-test split

train\_rows <- sample(1:nrow(df), nrow(df)\*0.8, replace = F)

train <- df[train\_rows, ]

test <- df[-train\_rows, ]

#####

# Build model

inputs <- layer\_input(shape = c(2))

outputs <- inputs %>%

layer\_dense(8, activation = 'elu') %>%

layer\_dense(4, activation = 'elu') %>%

layer\_dense(2, activation = 'linear')

model <- keras\_model(inputs = inputs, outputs = outputs)

model %>% compile(

optimizer = 'adam',

loss = 'mean\_absolute\_error',

metrics = c('mean\_squared\_error'))

model %>% fit(

as.matrix(train[1:2]),

as.matrix(train[4:5]),

validation\_data = list(

as.matrix(test[1:2]), as.matrix(test[4:5])),

epochs = 2000,

callbacks = list(

callback\_early\_stopping(

monitor = 'val\_loss',

mode = 'min',

patience = 150)))

# Test model

y\_pred <- as.data.frame(model %>% predict(as.matrix(test[1:2])))

print("MLP results:")

print(paste("MAE for cos(y) and sin(y) =", mean(abs(as.matrix(test[4:5] - y\_pred)))))

real\_y\_pred = atan2(y\_pred[,2], y\_pred[,1]) %% (2\*pi)

errors <- abs(test$y - real\_y\_pred)

# Correct for predictions near 2pi

errors <- if\_else(errors > pi, abs(errors - 2\*pi), errors)

print(paste("MAE for y =", mean(errors)))

**K-Nearest-Neighbors (To Compare)**

# R script

library(caret)

# Compare to a very simple model

knn\_model <- knnreg(train[1:2], as.matrix(train[4:5]), k = 2)

knn\_pred <- predict(knn\_model, as.data.frame(test[1:2]))

print("\nKNN results:")

print("MAE for cos(y) and sin(y) =", mean(abs(test$y - knn\_pred)))

real\_knn\_pred = atan2(knn\_pred[,2], knn\_pred[,1]) %% (2\*pi)

errors <- abs(test$y - real\_knn\_pred)

# Correct for predictions near 2pi

errors <- if\_else(errors > pi, abs(errors - 2\*pi), errors)

print("MAE for y =", mean(errors))

# Python script

from sklearn.neighbors import KNeighborsRegressor as KNN

# Compare to a very simple model

knn\_model = KNN(n\_neighbors=2)

knn\_model.fit(x\_train.reshape(-1, 2), y\_train)

knn\_pred = knn\_model.predict(x\_test.reshape(-1, 2))

print("\nKNN results:")

print("MAE for cos(y) and sin(y) =", np.mean(np.abs(y\_test - knn\_pred)))

real\_knn\_pred = np.mod(np.arctan2(knn\_pred[:,1], knn\_pred[:,0]), 2\*np.pi)

print("MAE for y =", np.mean(np.abs(real\_y\_test - real\_knn\_pred)))

**Results**

MLP results:

MAE for cos(y) and sin(y) = 0.056096028534601605

MAE for y = 0.08474142088002712

KNN results:

MAE for cos(y) and sin(y) = 0.06865971070310278

MAE for y = 0.10670556797752576

As shown by the below plot, the neural network tends to predict along the circle. It appears that using MSE for the loss function works well enough.

**A picture containing kite, flying, map, sky

Description automatically generated**

**Computing Distance**

Many algorithms rely on distance measurements. These include K-Nearest Neighbors and clustering algorithms like DBSCAN and K-Means. The distortions from using the trigonometric transformation described in the previous section are suboptimal for most of these algorithms. A type of cosine similarity can work but comes with its own issues that will be demonstrated later.

A flawless and simple solution for computing distances involving periodic attributes is to use the shortest distance along each periodic attribute to the neighboring point. Aside from attribute scaling, this is all that is needed because most if not all distance metrics use the differences along each dimension independently; for example, the Pythagorean theorem takes the square root of the sum of the squared differences to find the Euclidean distance.

A picture containing object

Description automatically generated

Precise implementation will vary, but the approach is to take the absolute difference between the points along the dimension and if the dimension is periodic and the difference is greater than half the length of the period, subtract the length of the period and return the absolute value.

It can be expensive to constantly compute the distance between a point and all others, especially when conditional checks are involved. One way to reduce computation is to compute a full distance matrix for later reference. Not only does the matrix only need to be computed once, but math on matrices can be parallelized. Precomputed distance matrixes are often the only way to supply a custom distance metric to a library function. Supplied below are functions written in R and Python for building a distance matrix when periodic attributes are involved:

# R Script

# Returns distance matrix ("dist" class) for data points, when some values belong to periodic/cyclical attributes.

# E.g. 23:59 is only one minute away from 00:00, not 23 hours and 59 minutes.

periodic\_dist <- function(dat,

ranges = NULL,

diag = T, upper = T) {

# Check arguments

if (sum(class(dat) %in% c("data.frame", "tbl\_df", "tbl")) < 1) {

stop("\nArgument 'dat' should be a tibble or dataframe.")

} else {

# This shouldn't happen, but if I can make the mistake someone else might.

if (sum(class(dat[[1]]) %in% c("data.frame", "tbl\_df", "tbl")) > 0) {

stop("\nIt appears you supplied a tibble containing a dataframe or tibble.\n",

"Did you use tibble() instead of as\_tibble()? Try nameOfYourData[[1]].")

}

}

if (is.null(ranges)) {

warning("\nArgument 'ranges' not provided.\n",

"All columns treated as periodic and set column ranges to lowest and highest values.")

ranges <- list()

for (col in dat) {

# Append vector to list

ranges <- c(ranges, list(range(col)))

}

} else {

if (class(ranges) != "list") {

stop(paste("\nProvide argument 'ranges' as a list of vectors and values describing each column.",

"If a column is periodic provide a min and max vector, like c(1, 12) for month, or a single range value (computed by min - max), like 2\*pi for angle.",

"If a column is not periodic, provide NA.",

"Example: list(c(0,1), c(1,12), 2\*pi, NA)",

sep = "\n"))

}

}

# Initialize distance matrix

distances <- dist(rep(0, nrow(dat)), diag = diag, upper = upper)

for (i in 1:length(dat)) {

col\_dist <- dist(dat[i], diag = diag, upper = upper)

if (sum(is.na(ranges[[i]])) < 1) {

# Replace values greater than half the possible distance

# with the remaining distance

range\_val <- ranges[[i]]

if (length(range\_val) > 1) {

range\_val <- range\_val[2] - range\_val[1]

}

too\_high <- which(col\_dist > range\_val/2)

col\_dist[too\_high] <-

abs(range\_val - col\_dist[too\_high])

}

# cat("col\_dist", col\_dist)

# Increment distance^2

distances <- distances + col\_dist^2

# cat("\n", distances)

}

# Complete distance formula by taking square root

return(sqrt(distances))

}

# Python script

import numpy as np

from scipy.spatial import distance\_matrix

import pandas as pd

import warnings

"""

dat:

A pandas dataframe or object that can be converted into pandas dataframe

ranges:

A list describing each column using any of these elements:

Float or integer that indicates the length of the period.

A list or tuple containing the min and max values of the period.

None. Indicates that the column is not periodic.

"""

def periodic\_distance\_matrix(dat, ranges = None):

if type(dat) != pd.core.frame.DataFrame:

dat = pd.DataFrame(dat)

if type(ranges) == type(None):

warnings.warn("No range information supplied. " +

"Taking max-min for each column.")

ranges = []

for col in dat:

ranges.append(np.ptp(dat[col]))

else:

if type(ranges) != list:

raise ValueError("Argument 'ranges' needs to be of type list.")

ranges = [np.ptp(r) if type(r) == list or type(r) == tuple

else r for r in ranges]

print(ranges)

distances = np.zeros((dat.shape[0],dat.shape[0])).astype('float64')

for i in range(dat.shape[1]):

values = dat.iloc[:,i].values.reshape(len(dat), 1)

col\_dist = distance\_matrix(values, values)

if ranges[i] != None:

col\_dist[col\_dist > ranges[i]/2] -= ranges[i]

distances += col\_dist\*\*2

return np.sqrt(distances)

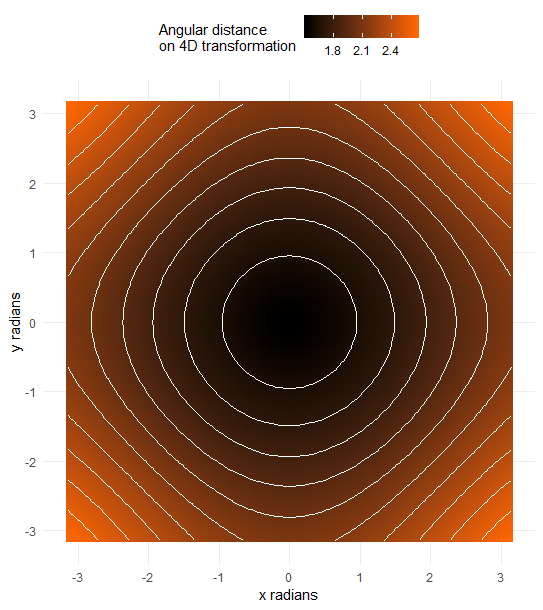
**What Ifs**

*Why only two?*One might choose to use more than two transformations. For example, three evenly spaced trigonometric transformations are sin(x), sin(x+2π/3), and sin(x+4π/3). This has no benefit as it creates a parameterized circle in three dimensions instead of two, which is subject to the same distortions at the cost of greater complexity.

*Can you just take the dot product to compare similarity?*  
If all or most variables are periodic, one might consider computing a type of angular distance as described:

This works better on trigonometrically transformed periodic variables than Euclidean distance does on the transformed variables, but it is still subject to its own distortions and is not helpful in situations where the distance must be precisely measured. The problem arises in the fact that the wraparound space created by looping rectangular spaces does not form proper hypersphere. This pseudo-angular distance metric could be more convenient in some cases or bring marginal calculation speed benefits, but this is still an inferior approach to the conditional algorithm provided previously because it doubles the number of dimensions to do an otherwise simple task. The graphics below show the behavior of the Euclidean (without trigonometric transformation, using an IF statement) and pseudo-angular (with transformation) distance measurements:

A close up of a logo

Description automatically generated

Note: The spacing between contour lines is not precisely the same between the two plots. There’s something with geom\_contour I would need to correct.

If two periodic variables are to be continuously expressed in three dimensions, it can only be done using a torus. An undistorted mapping can exist in 4D space. As mentioned, the shapes created by these wraparounds are not n-spheres. Some attempt has been made to demonstrate the four-dimensional shape.

In this 3D plot, the fourth dimension, w, is represented by the size and color of the points:A close up of a logo

Description automatically generated

After searching, I found that this is called a Clifford torus, belonging to a family of tori in even-numbered spaces. A proof for the Euclidean distance can be found here: <https://math.stackexchange.com/questions/965654/distance-between-two-points-on-the-clifford-torus>

Articles that may be of interest:

DOI: 10.1142/S0218001409007338