# Introduction to Artificial Neural Networks - Part 1: Multilayer Perceptron

## Introduction

This article is focussed on introducing some alternative machine learning concepts. The previous articles in the series have focussed on classical machine learning (ML) algorithms and techniques. Those original articles touched upon the basics of the machine learning process for supervised learning. They covered details about:

1. Exploratory data analysis and visualization.
2. Data hygiene and data preparation for model training.
3. Selecting popular algorithms to apply to regression and classification problems.
4. Comparable metrics for measuring the performance of models on test datasets.
5. Hyperparameters and optimization approaches to improve predictive performance.

Due to the breadth of ML algorithms and depth of research work done on tuning their performance, only a fraction of the use cases were examined to understand the overall process. The process steps given above are foundational and will be applied to any new algorithms that are going to be used for ML problems. Typically, steps 3 and 5 change together because the algorithm selection defines the parameters that can be tuned. Combining metrics examined in step 4 with the optimizations in step 5 create an iterative series of steps to improve performance. Selected algorithms demonstrated approaches to solutions in different problem spaces.

This article follows the same process defined above, but introduces a different algorithm class grouped under the heading of neural networks (NN) or artificial neural networks (ANN). There is a very large body of research available on ANN and its extension into deep NN (or deep learning) [9]. This area continues to grow, new techniques and architectures continue to be added, and its application to problems gets wider. In an introductory article, it is impossible to cover great depth or even touch upon all the different NN algorithms and approaches to use them for regression and classification problems.

This article will give exposure to the architecture, constraints and typical hyperparameters associated with NN. Surveying all the popular NN libraries and frameworks (e.g. TensorFlow or PyTorch or others) is outside the scope of this current article. This article will start with sklearn's multilayer perceptron library as the simplest demonstration of ANN.

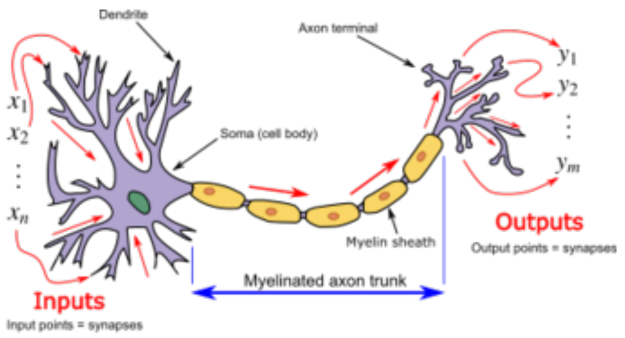
As with previous articles, the bulk of the discussion herein closely follows [12]. The primary purpose is to give the reader an overview of the details found throughout the reference and specifically in its chapter five. The datasets introduced in the reference will be explored and analyzed for doing the supervised learning of the ANN models. The experimental results will closely follow the exercises to demonstrate model functionality and its predictive capability on test data. The reader is encouraged to replicate these results and examine the source reference for additional details. This will provide an educational opportunity to gain practice using ANN, their main structures and features that can be optimized, and how to measure NN performance.

## Theory

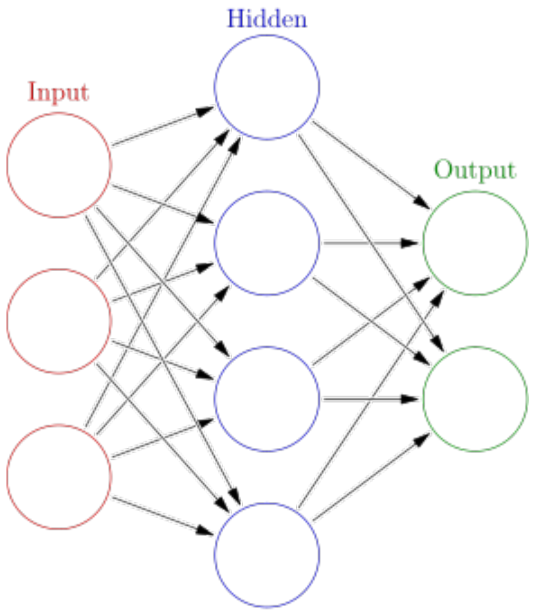
There is a lot of material published [1], [2], that covers how to utilize NN. Much of the theory behind MLP and NN is covered in a variety of sources (e.g. [3], [4], [5]), including the library's source code [6]. The MLP has its own long history [7], and its origins from the long development of the single layer perceptron [8].

NN gained popularity when their results on image classification problems began to match and then exceed human capacity. NN can be applied to a variety of domains. Other human actions being matched by NN include, speech recognition, natural language processing, and computer vision. More recent advances have found NN solving problems in modeling and prediction (e.g. finance, weather, medicine), fraud detection, recommendation, quality control and robotic autonomous systems. Truly new architectures and highly scaled NN (DNN, RNN, Transformers) have made contributions to generative image, sound, and text applications. These concepts are outside of scope of this article.

A common figure [10] showing the biological inspiration behind the theory of ANN is shown below. The basic idea is a neuron receives a series of input signals, which are carried forward to a termination point. If the signals sum up to a sufficient potential, then they cause an activation of an output signal to the next neuron in the chain. This causes the forward propagation of information through a series of neurons that ultimately recognize the input.



The more classical abstraction of the multiple layers of an MLP is shown [11] below. This typically represents the same features and functions as the biological neuron. The inputs are the feature set observed, the hidden neurons in the middle layer represent the units involved in detecting features and the output neurons represent the final decision of prediction. The MLP is most often used in classification tasks (binary or multi-class), but can be used in regression tasks too. The sklearn library implements algorithms for both tasks.

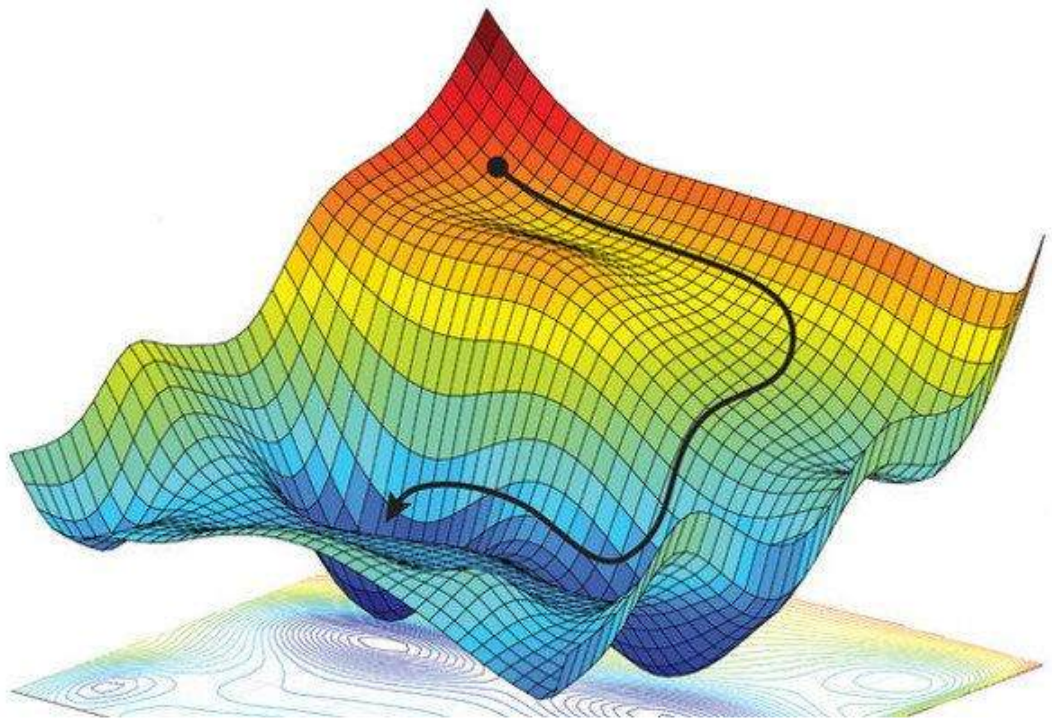


Repeating the fine details of the evolution from a single layer perceptron through the MLP, would make this article unpractically long. Instead the abstracted multiple layers figure will serve as a simple guide in the following discussion.

The basic idea behind NN is for each neuron (unit) to serve a specialized purpose, very much rooted in the biologic equivalents. That specialized purpose is typically recognizing some kind of foundational feature. If that feature is recognized, then that particular unit is considered to have satisfied its function by activating positively. This activation contributes to the overall identification (predicition) of the totality of features that represent the discrete classification target variable or the continuous regression target varible.

The steps involved in getting neurons to recognize foundational features define the supervised model training process. Assuming a cleaned dataset split into train and test sets is available for use, the data moving (forwards and backwards) through the MLP during training. This process involes forward propagation through the units at each layer, activiation functions which trigger the flow of information forward, and a process of backpropagation through the hidden layers which attempt to optimize the encoding of recognizing a feature by the layers of the NN. Like all optimization steps, there has to be a utility or cost function that must be maximized or minimized. One used in NN is the cross-entropy cost function. The opimiziation is done during backpropagation using the stochastic gradient descent (SGD) algorithm [13]. This SGD step is internally complicated [14], but is involved in updating the weights and biases of the equations representing the feature detection and encoded into the neurons.

A very common pictorial description of the SGD [20] algorithm's operations is demonstrated with a search over a surface in the direction of steepest descent. The goal being to find the global minima and not stuck in a local minima.



A more detailed and entertaining explination of SGD can be found in [15]. It covers pictorially some key aspects of the algorithm, its history, and the tunable hyperparameters involved. More importantly, it motivates when this particular algorithm can be applied and some of the problems associated with global and local minima.

Generally, NN described so far have a particular physical architecture. The physical components (for example, the quantity of hidden layers and neurons per layer) of the architecture are involved in the training process. Furthermore, the training action is composed of different components that include cost functions, activiation functions, the SGD algorithm, and their configuration (for example, the number of iterations to run, the maximum number of iterations, the learning rate). Lastly, the input data set and how it is consumed (for example, varying batch sizes or regularization of neurons to process the inputs) contribute to the functionality of the NN.

All of these architecture, algorithm and data features are considered hyperparameters of the MLP. As discussed in the introduction, the metrics used to measure prediction will quantify the performance of the NN. The NN performance can be improved by tuning the hyperparameters to produce higher metric scores. There is additional dicussion and background refresher material in the appendix.

## Analysis

### Datasets and Feature Engineering

This section examines the datasets involved in the experiments. It will go through two new datasets not seen in previous articles. The first dataset is on census income [16], [17] and the second dataset is on fertility diagnosis [18], [19]. Details about their features and target variables is discussed in the references.

These two datasets will be cleaned up prior to being used for training and test. The same techniques used in previous datasets will be applied. Additionally, the data will be split between training, test, and validation. An explanation of what a validation dataset is and the purpose of using it will be given.

Generically, the scikit-learn API is broken up into three main parts. A) Estimator B) Predictor C) Transformer. The three different parts of the API are utilized for the different functions involved in ML tasks. These ML tasks can be achieved using traditional ML algorithms or NN algorithms and architectures. Part A is used to fit models to training data. Part B extends models to do predictions on unseen test data. Part C is for data preprocessing techniques like scaling or standardization of training data.

Most of the features of the API have been used throughout the article series. These articles on ANN will also utilize many of the same functions. The dataset cleaning starts with some of the more common preprocessing techniques. Many times this is required due to the algorithms being used and how they can or can not interpret some data values. This is sometimes referred to as 'feature engineering' (FE).

Examples of FE can include, transforming the target variable to handle nominal types (text with no ranking) or ordinal types (text with ranking). This can be done with the LabelEncoder() API call for creating numeric values. Rescaling values is another technique that can be applied using A) normalization and B) standardization.

For Part A, normalization, the values are rescaled so they fall between [0,1] (inclusive) and have a max length of 1. For Part B, standardization, the data is transformed into a Gaussian distribution with a zero mean and standard deviation of one. With a mean value of 0, there can be both positive and negative values.

These FE operations are needed so the algorithms can use more homogenous scaling amongst all the features. This makes it easier to discover patterns, affects the model's performance, and allows for more equal weights to be used. This step is also complicated by experimentation, application and then evaluation to see if the operations used are effective in improving the algorithm's performance.

The concept of hyperparameter tuning as an optimization step was discussed earlier. It is typically based on the performance of the model on a test dataset. This test dataset is usually a percentage of the original dataset (20 or 30%) that is excluded from being used in model training (the remaining 70 or 80%). A better approach is to use what is termed the validation dataset for hyperparameter tuning steps.

Data is usually split into a training and test dataset as described earlier. This time, the training dataset is split again into a validation set and a training set. This validation set's purpose is to look like the test set, but it is not used in actually training the model. This could be achieved with a split of 60%-20%-20% (training-validation-test). The initial split might be 60% training and 40% test. The 40% test is split again in half into 20% validation and 20% test. More hyperparameters might require more validation set data in order to tune them.

The reason for doing this is to have the bulk of the model trained on the training set. The hyperparameter tuning requires extra data, so it is best to use data that has not been seen before and will not be part of the test set. Otherwise, there is a risk the model will overfit on the training dataset and not perform well on truly new and unseen data.

The concept of cross-validation was introduced in previous articles. It is used to partition data by resampling the data used to train and validate the model. It uses a parameter k (also known as the number of folds) to represent the resampling quantity.

Lastly, the traditional evaluation metrics are employed for ANN. In the case of classification, the confusion matrix and classification report (containing accuracy, precision and recall) are used. In the case of regression, the mean absolute error and root mean squared error are used. These metrics will appear as the outputs of the trained NN when doing prediction on the test data.

## Data Management

This section examines the datasets described in the previous section. Some basic information about the datasets will be examined to get an idea of what the data actually looks like. Additionally, it will also lend itself to some visualization and descriptive statistics.

### Fertility Dataset

The Fertility dataset at the link provided gives details about the nine features and one target variable in the CSV file. The source site indicates the data is complete and does not require additional cleaning actions. Additionally, the feature set data has already been encoded into numeric quantities. Only the target variable (column 9) remains as text, with values of: (N) for normal fertility and (O) for altered fertility. This allows for the data to be read into a dataframe prior to being used in training the NN. Column headers are excluded in the CSV files, but are added back in to display with the values.

Let's examine some basic information, correlation details, and visualize with a heatmap. The basics of its size (100 samples) and data types of each field are given by the dataframe's info() method. It very quickly gives the basic dimensions (shape) and if any data is missing from the rows.

In [20]:
```python
import pandas as pd
import numpy as np
```

In [21]:
```python
# this dataset does not contain any header rows.
# They are detailed on the website to give the 9 features.
fertility_diag_df = pd.read_csv("fertility_Diagnosis.csv", header=None)
# to get an idea of what the
fertility_diag_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       100 non-null    float64
 1   1       100 non-null    float64
 2   2       100 non-null    int64  
 3   3       100 non-null    int64  
 4   4       100 non-null    int64  
 5   5       100 non-null    int64  
 6   6       100 non-null    float64
 7   7       100 non-null    int64  
 8   8       100 non-null    float64
 9   9       100 non-null    object 
dtypes: float64(4), int64(5), object(1)
memory usage: 7.9+ KB
```

In [22]:
```
# some descriptive statistics
fertility_diag_df.describe()
```

Out[22]:

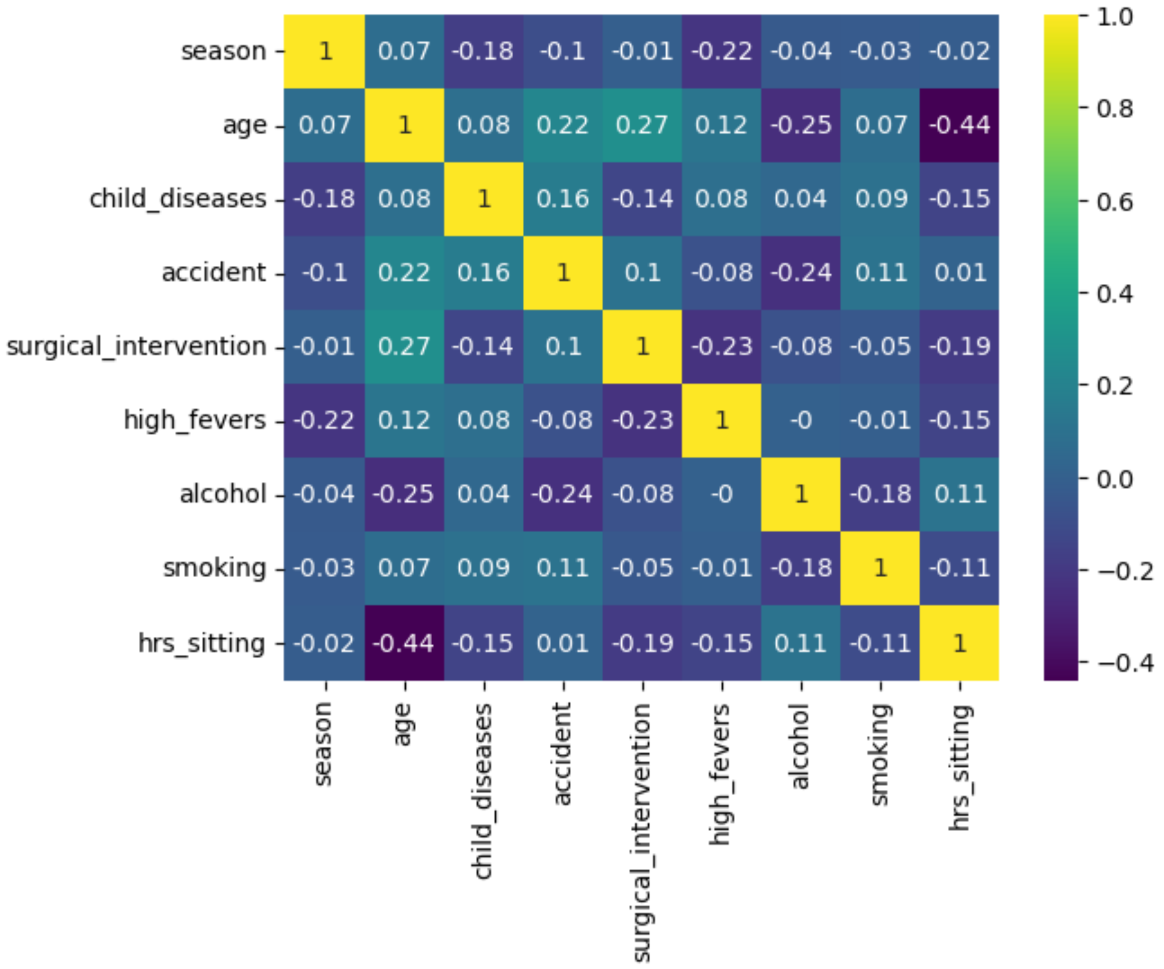
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 100.000000 | 100.000000 | 100.000000 | 100.000000 | 100.000000 | 100.000000 | 100.000000 | 100.000000 | 100.000000 |
| **mean** | -0.078900 | 0.669000 | 0.870000 | 0.440000 | 0.510000 | 0.190000 | 0.832000 | -0.350000 | 0.406800 |
| **std** | 0.796725 | 0.121319 | 0.337998 | 0.498888 | 0.502418 | 0.580752 | 0.167501 | 0.808728 | 0.186395 |
| **min** | -1.000000 | 0.500000 | 0.000000 | 0.000000 | 0.000000 | -1.000000 | 0.200000 | -1.000000 | 0.060000 |
| **25%** | -1.000000 | 0.560000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.800000 | -1.000000 | 0.250000 |
| **50%** | -0.330000 | 0.670000 | 1.000000 | 0.000000 | 1.000000 | 0.000000 | 0.800000 | -1.000000 | 0.380000 |
| **75%** | 1.000000 | 0.750000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 0.000000 | 0.500000 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

In [23]:
```
# Set the column headings for display purposes only.
fertility_diag_df.columns = ['season', 'age', 'child_diseases', 'accident', 'surgical_intervention', 'high_fevers', 'alcohol', 'smokir
fertility_diag_df.head()
```

Out[23]:

| | season | age | child_diseases | accident | surgical_intervention | high_fevers | alcohol | smoking | hrs_sitting | diagnosis |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | -0.33 | 0.69 | 0 | 1 | 1 | 0 | 0.8 | 0 | 0.88 | N |
| **1** | -0.33 | 0.94 | 1 | 0 | 1 | 0 | 0.8 | 1 | 0.31 | O |
| **2** | -0.33 | 0.50 | 1 | 0 | 0 | 0 | 1.0 | -1 | 0.50 | N |
| **3** | -0.33 | 0.75 | 0 | 1 | 1 | 0 | 1.0 | -1 | 0.38 | N |
| **4** | -0.33 | 0.67 | 1 | 1 | 0 | 0 | 0.8 | -1 | 0.50 | O |

In [24]:
```python
# examine the data being correlated to each other and attempt to display it.
import seaborn as sns
import matplotlib.pyplot as plt
# calculate the correlation info into a dataframe
fertility_diag_df_corr = fertility_diag_df.corr(numeric_only=True).round(2)

sns.heatmap(fertility_diag_df_corr, annot=True, fmt="g", cmap='viridis')
plt.show()
```



| | season | age | child_diseases | accident | surgical_intervention | high_fevers | alcohol | smoking | hrs_sitting |
|---|---|---|---|---|---|---|---|---|---|
| season | 1 | 0.07 | -0.18 | -0.1 | -0.01 | -0.22 | -0.04 | -0.03 | -0.02 |
| age | 0.07 | 1 | 0.08 | 0.22 | 0.27 | 0.12 | -0.25 | 0.07 | -0.44 |
| child_diseases | -0.18 | 0.08 | 1 | 0.16 | -0.14 | 0.08 | 0.04 | 0.09 | -0.15 |
| accident | -0.1 | 0.22 | 0.16 | 1 | 0.1 | -0.08 | -0.24 | 0.11 | 0.01 |
| surgical_intervention | -0.01 | 0.27 | -0.14 | 0.1 | 1 | -0.23 | -0.08 | -0.05 | -0.19 |
| high_fevers | -0.22 | 0.12 | 0.08 | -0.08 | -0.23 | 1 | -0 | -0.01 | -0.15 |
| alcohol | -0.04 | -0.25 | 0.04 | -0.24 | -0.08 | -0 | 1 | -0.18 | 0.11 |
| smoking | -0.03 | 0.07 | 0.09 | 0.11 | -0.05 | -0.01 | -0.18 | 1 | -0.11 |
| hrs_sitting | -0.02 | -0.44 | -0.15 | 0.01 | -0.19 | -0.15 | 0.11 | -0.11 | 1 |

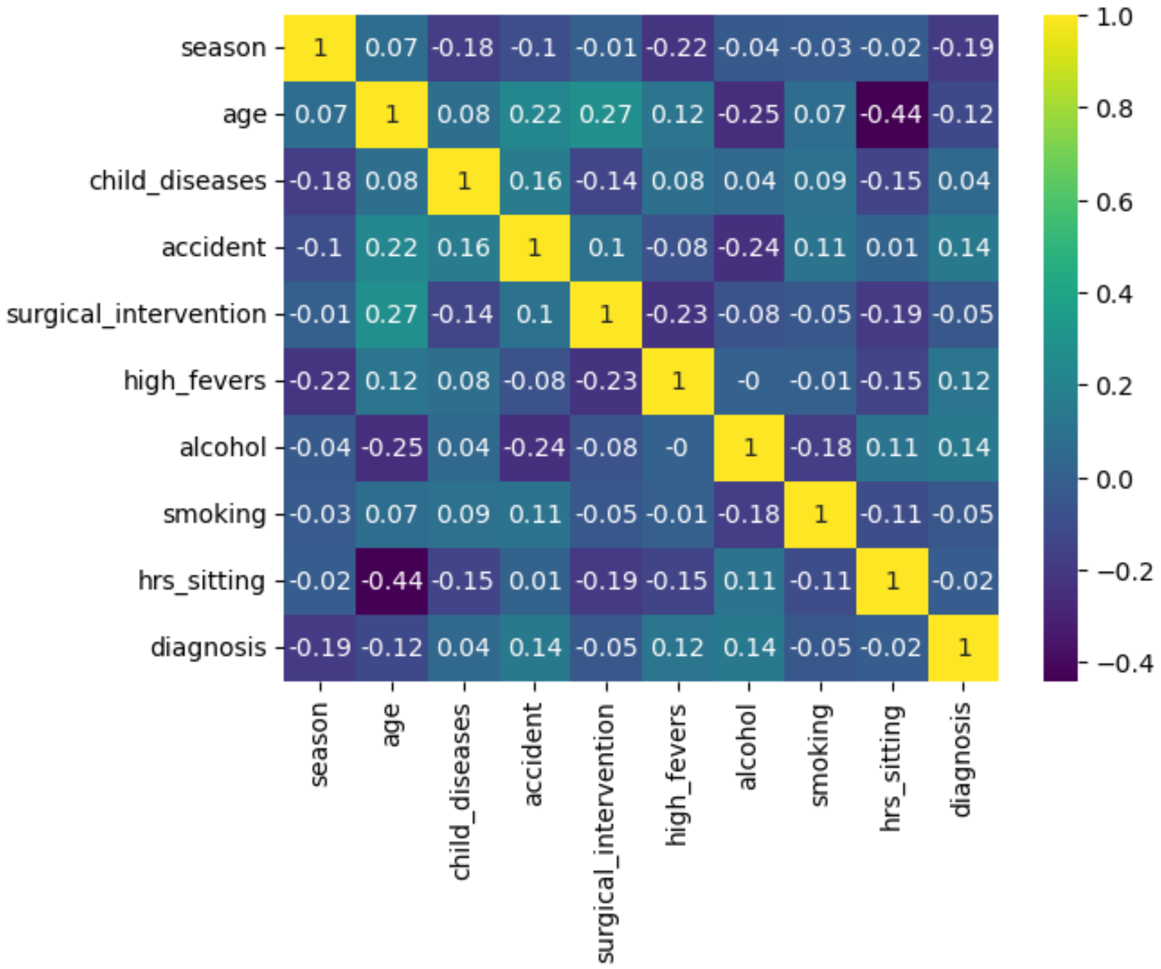
Notice the heatmap is missing the target variable because it is a character and can not be correlated to the numeric feature columns. Converting that last column to a numeric encoding will give an additional variable to plot and will show the correlation of the features with the target.

In [26]:
```python
# Update the target column values with 1 for Normal (N) and 0 for Altered (O)
fertility_diag_df_numeric = fertility_diag_df.replace(['N', 'O'], [1,0])
# examine what the new df looks like after the substitution
fertility_diag_df_numeric.head()
```

Out[26]:

| | season | age | child_diseases | accident | surgical_intervention | high_fevers | alcohol | smoking | hrs_sitting | diagnosis |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | -0.33 | 0.69 | 0 | 1 | 1 | 0 | 0.8 | 0 | 0.88 | 1 |
| **1** | -0.33 | 0.94 | 1 | 0 | 1 | 0 | 0.8 | 1 | 0.31 | 0 |
| **2** | -0.33 | 0.50 | 1 | 0 | 0 | 0 | 1.0 | -1 | 0.50 | 1 |
| **3** | -0.33 | 0.75 | 0 | 1 | 1 | 0 | 1.0 | -1 | 0.38 | 1 |
| **4** | -0.33 | 0.67 | 1 | 1 | 0 | 0 | 0.8 | -1 | 0.50 | 0 |

In [27]:
```
# calculate the correlation info into a dataframe
fertility_diag_df_numeric_corr = fertility_diag_df_numeric.corr(numeric_only=True).round(2)
# display the new heatmap
sns.heatmap(fertility_diag_df_numeric_corr, annot=True, fmt="g", cmap='viridis')
plt.show()
```



This figure now shows which feature variables are most positively and negatively correlated to the diagnosis target variable. The features of accident, high fevers, and alcohol are most positively correlated.

No additional work needs to be done on this dataset. The experimenal results will operate on these dataframes to extract data for training the NN.

## Census-Income Dataset

The next dataset is the 1994 census dataset from the reference below. This dataset is much larger (32,561 samples, 14 features, 1 binary class label) and contains adult demographics data collected during the census. The discrete label, that is the target variable to predict, is if the person's income is greater than 50,000 dollars or less than or equal to 50,000 dollars.

The number of features is large and is not reproduced here. All the column headings, the type of the feature, and some relevant information is captured in the reference given. There is a mix of quantitative (all continuous) and qualitative (nominal and ordinal) values.

Starting with the imported CSV data file, the dataframe is cleaned up and displayed to determine the scope of the data that will be used for training. These operations again similarly rely on library methods to do most of the pre-processing and cleaning in preparation for model training.

In [31]:
```
# bring in the data set first into a dataframe.
census_income_df = pd.read_csv("adult_data.csv", header=None)
# and determine its basic info
census_income_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   0       32561 non-null  int64 
 1   1       32561 non-null  object
 2   2       32561 non-null  int64 
 3   3       32561 non-null  object
 4   4       32561 non-null  int64 
 5   5       32561 non-null  object
 6   6       32561 non-null  object
 7   7       32561 non-null  object
 8   8       32561 non-null  object
 9   9       32561 non-null  object
 10  10      32561 non-null  int64 
 11  11      32561 non-null  int64 
 12  12      32561 non-null  int64 
 13  13      32561 non-null  object
 14  14      32561 non-null  object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

In [32]:
```
# examining the data pulled in
census_income_df.columns = ['age', 'workclass', 'fnlwgt', 'education', 'education-num', 'marital-status', 'occupation', 'relationship'
census_income_df.head()
```

Out[32]:

| | age | workclass | fnlwgt | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | income |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | State-gov | 77516 | Bachelors | 13 | Never-married | Adm-clerical | Not-in-family | White | Male | 2174 | 0 | 40 | United-States | <=50K |
| **1** | 50 | Self-emp-not-inc | 83311 | Bachelors | 13 | Married-civ-spouse | Exec-managerial | Husband | White | Male | 0 | 0 | 13 | United-States | <=50K |
| **2** | 38 | Private | 215646 | HS-grad | 9 | Divorced | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 40 | United-States | <=50K |
| **3** | 53 | Private | 234721 | 11th | 7 | Married-civ-spouse | Handlers-cleaners | Husband | Black | Male | 0 | 0 | 40 | United-States | <=50K |
| **4** | 28 | Private | 338409 | Bachelors | 13 | Married-civ-spouse | Prof-specialty | Wife | Black | Female | 0 | 0 | 40 | Cuba | <=50K |

In [33]:

```
# descriptive statistics.  Just a few features are shown because they are the default numeric ones.
# Other qualitative features will be converted to quantitative values for use in training.
census_income_df.describe()
```

Out[33]:

| | age | fnlwgt | education-num | capital-gain | capital-loss | hours-per-week |
|---|---|---|---|---|---|---|
| **count** | 32561.000000 | 3.256100e+04 | 32561.000000 | 32561.000000 | 32561.000000 | 32561.000000 |
| **mean** | 38.581647 | 1.897784e+05 | 10.080679 | 1077.648844 | 87.303830 | 40.437456 |
| **std** | 13.640433 | 1.055500e+05 | 2.572720 | 7385.292085 | 402.960219 | 12.347429 |
| **min** | 17.000000 | 1.228500e+04 | 1.000000 | 0.000000 | 0.000000 | 1.000000 |
| **25%** | 28.000000 | 1.178270e+05 | 9.000000 | 0.000000 | 0.000000 | 40.000000 |
| **50%** | 37.000000 | 1.783560e+05 | 10.000000 | 0.000000 | 0.000000 | 40.000000 |
| **75%** | 48.000000 | 2.370510e+05 | 12.000000 | 0.000000 | 0.000000 | 45.000000 |
| **max** | 90.000000 | 1.484705e+06 | 16.000000 | 99999.000000 | 4356.000000 | 99.000000 |

For this analysis, these 5 features: fnlwgt, education, relationship, race, sex are deleted. There are 5 additional qualitative features that require conversion to numerics. These can be encoded using the LabelEncoder library. The info() method will be examined to see how many excess null values appear in different columns. Lastly, excess outliers (datapoints in excess of 3 standard deviations) will be considered for removal.

In [35]:

```
# drop these columns (these include ordinal qualitative features, that must be converted carefully in comparison to nominal values)
census_income_df.drop(["fnlwgt", "education", "relationship", "race", "sex"], axis=1, inplace=True)
```

Next create a function to handle the encoding of qualitative features.

In [37]:

```
from sklearn.preprocessing import LabelEncoder
def encode_qual_features(dfToCheck, qualFeatureList):
    ''' this function takes the arguments of the dataframe to be checked
    and the list of qualitative features to be encoded '''
    # convert these with encoding
    df_encoder = LabelEncoder()
    for i in qualFeatureList:
        dfToCheck[i] = df_encoder.fit_transform(dfToCheck[i].astype('str'))
```

In [38]:
```
# the examination of data earlier showed the qualitative data features to convert
convert_features = ["workclass", "marital-status", "occupation", "native-country", "income"]
encode_qual_features(census_income_df, convert_features)
# display what the new dataframe looks like now
census_income_df.head()
```

Out[38]:

| | age | workclass | education-num | marital-status | occupation | capital-gain | capital-loss | hours-per-week | native-country | income |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 39 | 7 | 13 | 4 | 1 | 2174 | 0 | 40 | 39 | 0 |
| **1** | 50 | 6 | 13 | 2 | 4 | 0 | 0 | 13 | 39 | 0 |
| **2** | 38 | 4 | 9 | 0 | 6 | 0 | 0 | 40 | 39 | 0 |
| **3** | 53 | 4 | 7 | 2 | 6 | 0 | 0 | 40 | 39 | 0 |
| **4** | 28 | 4 | 13 | 2 | 10 | 0 | 0 | 40 | 5 | 0 |

Next, create a function to handle and examine outliers. If there are any features with outlier samples, the quantity of them must be considered to see the impact of their removal.

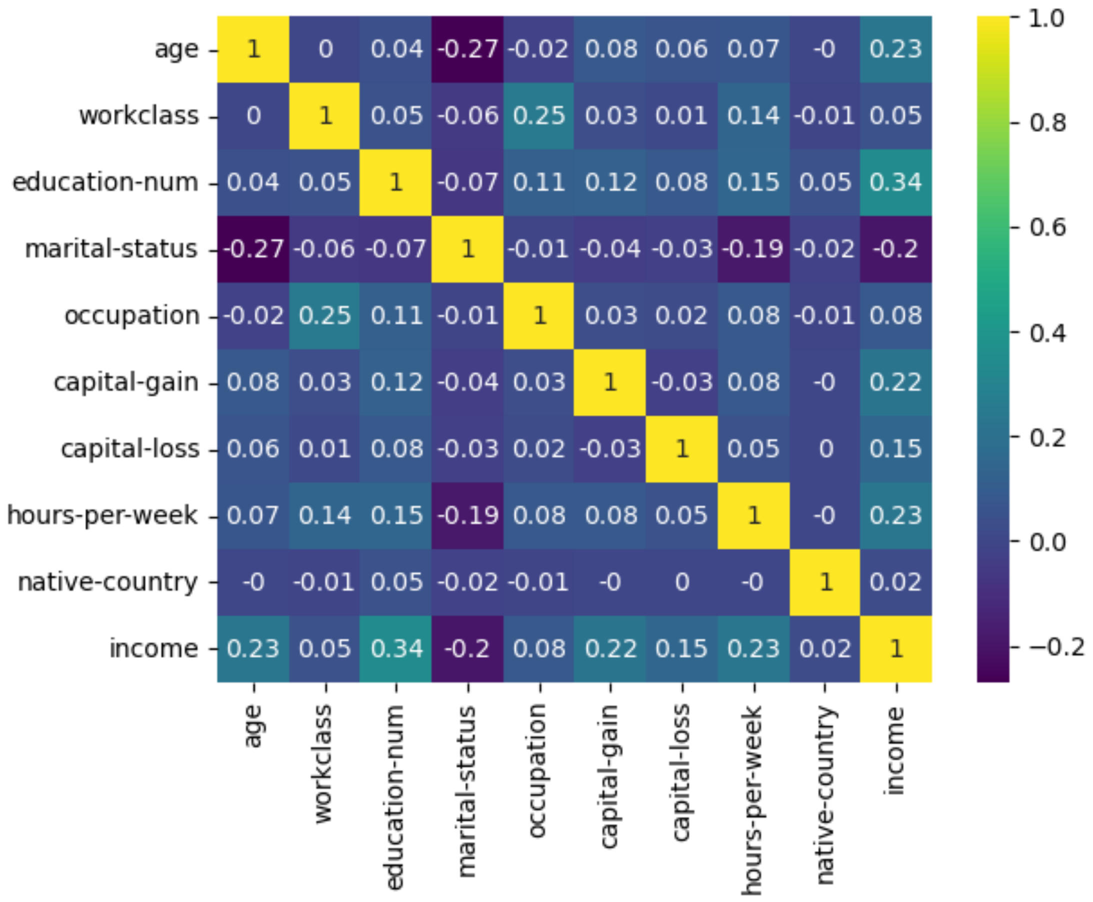
In [40]:
```
# check how many outliers exist that need to be handled using this function
def check_outliers(dfToCheck):
    ''' function takes in the dataframe with columns to be checked
    and prints out relevant calculations.  It attempts to see which
    rows have values that are in excess of 3 standard deviations
    to qualify as an outlier.  '''
    for i in range(dfToCheck.shape[1]):
        min_t = dfToCheck[dfToCheck.columns[i]].mean() - (3*dfToCheck[dfToCheck.columns[i]].std())
        max_t = dfToCheck[dfToCheck.columns[i]].mean() + (3*dfToCheck[dfToCheck.columns[i]].std())
        # establish a count of the number of values outside of these two ranges
        count = 0
        #
        for j in dfToCheck[dfToCheck.columns[i]]:
            if j < min_t or j > max_t:
                count +=1
        # done, collect data
        print([count, dfToCheck.shape[0]-count])
        print(f'{dfToCheck.columns[i]} outlier percentage: {((count/(dfToCheck.shape[0]-count))*100)}')
```

In [41]:
```
# test for outliers and determine those counts
check_outliers(census_income_df)
```

```
[121, 32440]
age outlier percentage: 0.37299630086313196
[0, 32561]
workclass outlier percentage: 0.0
[219, 32342]
education-num outlier percentage: 0.6771380866984107
[0, 32561]
marital-status outlier percentage: 0.0
[0, 32561]
occupation outlier percentage: 0.0
[215, 32346]
capital-gain outlier percentage: 0.6646880603474927
[1470, 31091]
capital-loss outlier percentage: 4.728056350712425
[440, 32121]
hours-per-week outlier percentage: 1.369820366738271
[1505, 31056]
native-country outlier percentage: 4.846084492529624
[0, 32561]
income outlier percentage: 0.0
```

Based on the counts calculated and data available, the quantity of outliers is not large enough to warrent any additional data removal. The largest one is under 5% and the others are very small. So no additional data needs to be cleaned out of this dataset.

In [43]:
```
# calculate the correlation info into a dataframe
census_income_df_corr = census_income_df.corr(numeric_only=True).round(2)
# display the new heatmap
sns.heatmap(census_income_df_corr, annot=True, fmt="g", cmap='viridis')
plt.show()
```



The features showing highest positive correlation to income include: age, education, capital-gain and loss, and worked hours-per-week. This basic analysis and visualization of the cleaned datasets has now prepared them for use in model training.

## Experimental Results

This section covers the process of training an MLP model. The datasets described earlier are used in the training process. The initial experiments will be with a network architecture using some number of layers with some number of neurons per layer with fixed training parameters.

Some observations of NN training include the following:

1. It is a time consuming process due to optimization convergence time.
2. NN require lots of data to find patterns (on the order of millions of datapoints for specific problems).
3. The training process yields values for weights and biases but how those values are arrived is probabilistic and not exactly known.
4. NN training is hardware intensive. Complex NN require more compute to train in a reasonable time period.

### Hyperparameters

There are many hyperparameters that can be tuned, but those will be changed in subsequent experiments to optimize the predictions (model accuracy versus error rates). Full details of all the hyperparameters can be found in the documentation of the algorithm source code. Not all hyperparameters have to be tuned everytime. A lot depends on dataset properties and how the model training is performing on the dataset.

Examples and explinations of some of the hyperparameters tuned in the experiments include:

1. The number of hidden layers
2. The number of units (neurons) per layer
3. Activation functions
4. Regularization techniques
5. Batch size
6. Learning rate
7. The number of iterations to run in training
8. The maximum number of iterations to run the opimization

The number of hidden layers and the number of units per layer does not follow a specific formula. Differing number of layers and neurons can be selected to train a network. In each case, the error statistics must be examined to determine how accurate the predictions are over the test dataset. Adjusting these quantities will produce different error rates to compare. A range of values might have to be tried in training multiple models in order to find the near-optimal set.

It is best to start with initial values associated with other similiar network architectures used to address datasets with similar properties. Generally, deeper networks (those with more layers) perform better than wider networks (those with many neurons per layer).

Activation functions are the next parameter that needs optimization. Some of the popular ones are discussed in the references cited. For the hidden layers, the ReLU (rectilinear unit) and hyperbolic tangent are used, with ReLU being preferred. For the output layer, the Sigmoid and Softmax are used to output a probability value. Sigmoid can be used in binary classification and Softmax can be used in binary or multi-class classification.

Regularlization (as discussed in the appendix) is used to treat model overfitting. It is used primarily to treat this specific symptom and is supposed to increase model generalizability. Examples of regularization include L1, L2, and dropout.

1. L1 (absolute values of the weight magnitudes) and L2 (squared magnitude of the weights) change the cost function. L2 is the primary one used.
2. Dropout is the removal of units during iteration to simplify the network. These units and their output values are ignored.

Batch sizes refers to how many instances (examples or rows) are put into a NN during an iteration. This parameter is used to help generalize overfitting models. Usual batch size values includes 32, 64, 128, 256.

Learning rate affects the step size used in training iterations to find the global minima. This concept was examined in the SGD descriptions. Lower learning rates slow the training process down but produce better results because it improves the chances of finding the minima. Larger learning rates speed up the training process, but optimization might not converge. An example rate can be set to 0.001.

The number of iterations is the number of forward and backward propagation passes to take. It is a number that can be adjusted from initial small values (200-500) to larger values. Training time increases as a function of iterations. The optimal number is found by plotting the cost function results to observe maximal decrease. Underfitting models can be improved by increasing this number.

## NN Training

These subsections examine MLP training on the two datasets shown and cleaned earlier. These will both be examined first to see how they perform on the datasets. Then some optimization approaches will be tried to improve performance.

In both cases, the MLP model and split libraries are required to do the necessary steps. These are brought in first.

In [51]:
```
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
```

### Fertility Dataset

In [53]:
```
# This section picks up on the cleaned dataset from earlier.
# The fertility dataset is very small and is not split up into training, validation and test sets.
# Just the model construction, fitting, and prediction on a sample input are shown.
```

In [54]:
```
# Create a function to handle the processing
def process_fertility_dataset(X, y):
    # note the max iterations are set high due to convergence error for default iterations of 200.
    model = MLPClassifier(random_state=101, max_iter = 1200)
    model = model.fit(X, y)
    # sample to predict using the first 9 feature values.  See column headers supplied earlier.
    sample_features = [-0.33, 0.69, 0,1,1,0,0.8,0,0.88]
    pred = model.predict([sample_features])
    print(f'model prediction = {pred}')
```

In [55]:
```
# The data is stored in: fertility_diag_df and is broken up into features and target for passing in.
# The first 9 columns are the features and the last column is the target variable
process_fertility_dataset(fertility_diag_df.iloc[:,:9], fertility_diag_df.iloc[:,9])
```

```
model prediction = ['N']
/home/ap/anaconda3/lib/python3.10/site-packages/sklearn/base.py:420: UserWarning: X does not have valid feature names, but MLPClassifi
er was fitted with feature names
  warnings.warn(
```

The output shown is N for normal. The MLP was trained on the supplied features first and then given a set of sample features to predict on. These features yield a result of normal.

### Census Income Dataset

The census_income_df was created in the analysis section. Its data was cleaned and some qualitative data was dropped. Lastly, some of the relevant (see the correlation heat map given earlier) qualitative data was converted into quantitative features for the MLP to process.

This dataframe will be used in the model the same way as the fertility dataset. A model will be created, then fit with training data, and then used on validation and test datasets to get accuracy scores. The cleaned dataframe contains the remaining nine features and one target feature. These will be separated out and supplied as the X matrix and y column vector.

In [59]:
```
# create a new function for error analysis
def error_analysis_census_dataset(accuracy_list, baseline_error_rate):
    # The baseline error rate is used to calculate the differences
    # amongst the different sets of data error rates to determine which
    # aspect of the model needs tuning
    # error rates = 1 - accuracy_score
    names = ['train', 'validate', 'test']
    previous_error = baseline_error_rate
    for i in range(0, len(accuracy_list)):
        error_rate = (1.0-accuracy_list[i]).round(4)
        print(f'{names[i]} accuracy score: {accuracy_list[i]}, error rate: {error_rate} difference: {(error_rate - previous_error).rou
        previous_error = error_rate
```

In [60]:
```
# define a function to manage all the steps described
def process_census_dataset(X, y, iterations, hidden_layers_tuple):
    # start by spliting the dataset up into train, validation, and test datasets
    X_new, X_test, y_new, y_test = train_test_split(X, y, test_size=0.1, random_state=101)
    # then take out the validation data from the training data by splitting again
    X_train, X_validate, y_train, y_validate = train_test_split(X_new, y_new, test_size=0.1111, random_state=101)
    # create the model
    # here, passing in additional hyperparameters that can be tuned by sending in different values.
    # the iterations is the maximum number of iterations to run the training for convergence
    # the hidden_layers_tuple describes the number of hidden layes and number of units per layer
    model = MLPClassifier(random_state=101, max_iter = iterations, hidden_layer_sizes=hidden_layers_tuple)
    model = model.fit(X_train, y_train)
    # now with the fitted model, want to get prediction accuracy across the three sets of data created
    # do this with a loop
    X_sets = [X_train, X_validate, X_test]
    Y_sets = [y_train, y_validate, y_test]
    # collect up the scores for analysis
    accuracy = []
    for i in range(0, len(X_sets)):
        pred = model.predict(X_sets[i])
        score = accuracy_score(Y_sets[i], pred)
        accuracy.append(score.round(4))
    # now display the scores, using a baseline error rate of 1% for comparison purposes
    error_analysis_census_dataset(accuracy, baseline_error_rate=0.01)
```

In [61]:
```
# want to split up the data first and then call function with it.
ci_df_X = census_income_df.drop('income', axis=1)
ci_df_y = census_income_df['income']
# pass in default values for iterations and hidden layers
process_census_dataset(ci_df_X, ci_df_y, 200, (100))
```

```
train accuracy score: 0.8379, error rate: 0.1621 difference: 0.1521
validate accuracy score: 0.8197, error rate: 0.1803 difference: 0.0182
test accuracy score: 0.8296, error rate: 0.1704 difference: -0.0099
```

## Hyperparameter tuning

The error analysis shows that the difference is greatest from a baseline error rate for the training dataset accuracy. This represents high bias (data underfitting) and its treatment (as described in the appendix) is to use a larger network or have a higher number of iterations. These hyperparameters can be tuned to improve model performance over this data.

This is accomplished by changing the two parameters passed into the model creation. A range of values are tried and the scores gathered to compare the performance of each experiment against the baseline default values supplied in the first instance. These are passed into the function above using the stored values in the dictionary.

In [64]:
```
# define the set of values to try in the experimental trial and then call each one.
iterations = [500, 500, 500, 500, 500, 500, 500]
# the length of each tuple is the number of hidden layers, and the value of
# each tuple entry is the number of units per layer
layers = [(100,), (100,100), (100,100,100), (50,50), (150,150), (25,25), (175,175)]
```

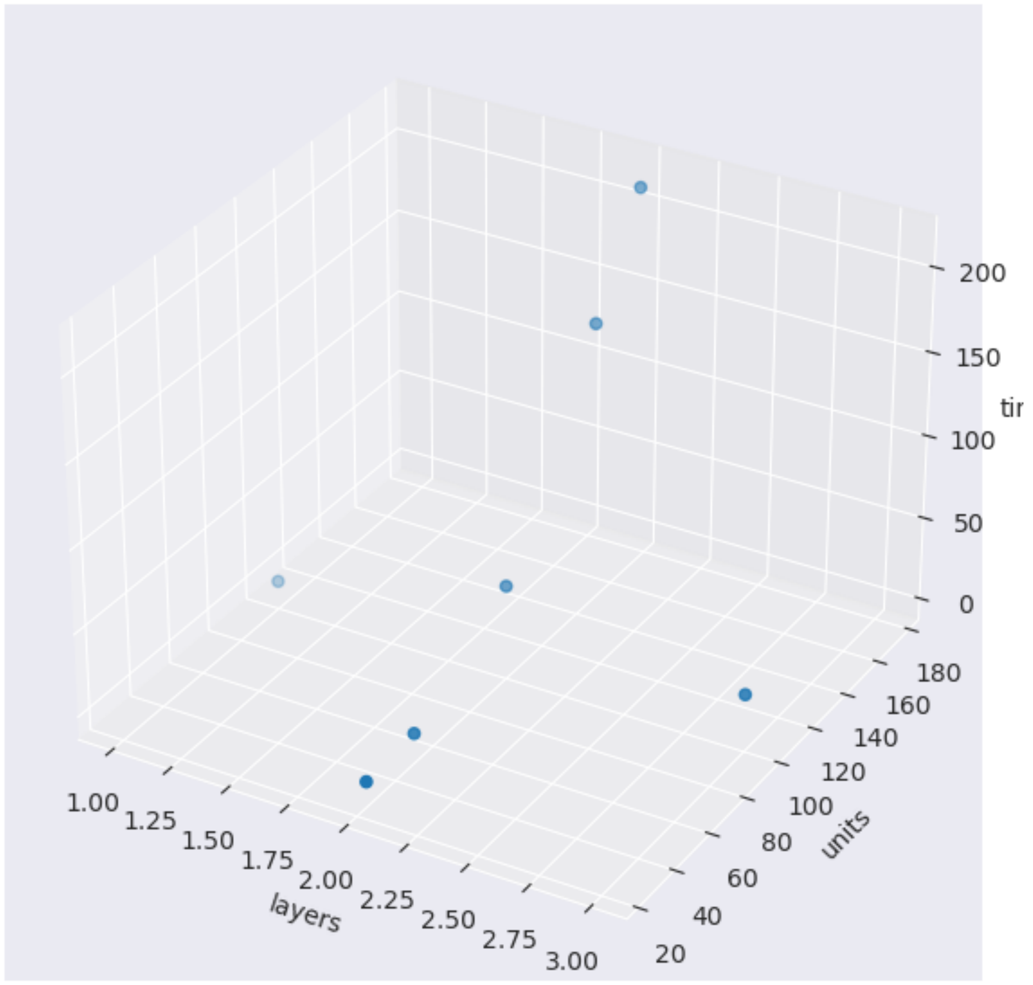
In [65]:
```
# Conduct the experiments trying different values for the hyperparameters to get the best scores.
import time

# collect up some data to plot.
# The iterations are all the same, so those do not need to be collected.
# Examine the plotting of the number of layers, the units per layer and training time.
number_of_layers = []
number_of_units_per_layer = []
training_time = []

for count in range(len(iterations)):
    start_time = time.perf_counter()
    itrs = iterations[count]
    lyrs = layers[count]
    process_census_dataset(ci_df_X, ci_df_y, itrs, lyrs)
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    #
    number_of_layers.append(len(lyrs))
    number_of_units_per_layer.append(lyrs[0])
    training_time.append(elapsed_time)
    print(f'\n#{count+1} iters={itrs} hidden layer(units/layer)={lyrs}, seconds to train {elapsed_time:.6f}\n')
```

```
train accuracy score: 0.8379, error rate: 0.1621 difference: 0.1521
validate accuracy score: 0.8197, error rate: 0.1803 difference: 0.0182
test accuracy score: 0.8296, error rate: 0.1704 difference: -0.0099

#1 iters=500 hidden layer(units/layer)=(100,), seconds to train 3.671232

train accuracy score: 0.8451, error rate: 0.1549 difference: 0.1449
validate accuracy score: 0.8188, error rate: 0.1812 difference: 0.0263
test accuracy score: 0.8391, error rate: 0.1609 difference: -0.0203

#2 iters=500 hidden layer(units/layer)=(100, 100), seconds to train 43.373380

train accuracy score: 0.824, error rate: 0.176 difference: 0.166
validate accuracy score: 0.8031, error rate: 0.1969 difference: 0.0209
test accuracy score: 0.8118, error rate: 0.1882 difference: -0.0087

#3 iters=500 hidden layer(units/layer)=(100, 100, 100), seconds to train 21.284076

train accuracy score: 0.8006, error rate: 0.1994 difference: 0.1894
validate accuracy score: 0.789, error rate: 0.211 difference: 0.0116
test accuracy score: 0.801, error rate: 0.199 difference: -0.012

#4 iters=500 hidden layer(units/layer)=(50, 50), seconds to train 4.676876

train accuracy score: 0.8666, error rate: 0.1334 difference: 0.1234
validate accuracy score: 0.8375, error rate: 0.1625 difference: 0.0291
test accuracy score: 0.8431, error rate: 0.1569 difference: -0.0056

#5 iters=500 hidden layer(units/layer)=(150, 150), seconds to train 154.861370

train accuracy score: 0.7765, error rate: 0.2235 difference: 0.2135
validate accuracy score: 0.7672, error rate: 0.2328 difference: 0.0093
test accuracy score: 0.7832, error rate: 0.2168 difference: -0.016

#6 iters=500 hidden layer(units/layer)=(25, 25), seconds to train 1.985522

train accuracy score: 0.8736, error rate: 0.1264 difference: 0.1164
validate accuracy score: 0.8326, error rate: 0.1674 difference: 0.041
test accuracy score: 0.8428, error rate: 0.1572 difference: -0.0102

#7 iters=500 hidden layer(units/layer)=(175, 175), seconds to train 214.105162
```

This tuning effort to run the model through a series of iterations and hidden layer/units-per-layer combinations gives different accuracy scores and error rates. Examining the different values for the scores shows that experiment #5 and #7 perform better than the others. Experiment #7 appears better in training and validation scores, but not the test scores. Experiment #5 outperforms in the test scores. Therefore, that experiment with 500 iterations and 2 hidden layers of 150 units each gives the best performance.

In [118…

```python
sns.set_style("darkgrid")
plt.figure(figsize=(7,8))
sns_plot = plt.axes(projection='3d')
sns_plot.scatter3D(np.array(number_of_layers), np.array(number_of_units_per_layer), np.array(training_time))
sns_plot.set_xlabel('layers')
sns_plot.set_ylabel('units')
sns_plot.set_zlabel('time')
plt.show()
```



Another interesting artifact to observe from the training output is the duration of time it took. The number of seconds to train is shown for each experiment. In general, and very roughly, the longer the training durations, the better the performance. It can not be stated that it will always be the case, but for these architectures defined by the hyperparameter values used, the longer duration model training gave better accuracy.

The plot also shows another aspect of training time (z-axis). In this very simple experiment, as the number of units increases, the training time increases. The number of hidden layers was not that impactful here. Many other combinations can be tried in tuning these two hyperparameters at the cost of run time. Additionally, hyperparameters mentioned before related to SGD can also be tuned.

Other metrics can also be examined and used to determine the quality of performance. Some might be more relevant to either the NN architecture or to the dataset being analyzed. In either case, they can be added to give a better idea of how to deal with tuning the NN to improve its prediction accuracy.

## Conclusion

This article has covered a number of different topics starting with the history of NN and the MLP to actually building small models to train. The technology behind NN has biological inspirations and its ideas are rooted in neuron functionality. The sklearn libraries available implement the MLP algorithm that provides a way to train a model on different dataset inputs.

This article also introduced two new datasets related to a fertility dataset and census income dataset. Feature engineering was demonstrated to manage the data for the purposes of cleaning it, for normalizing it, and converting appropriate columns of data to be usable by the NN. Lastly, the idea of hyperparameter tuning was introduced in the context of NN and the MLP algorithm. Hyperparameter tuning has been seen in previous ML algorithms that were demonstrated in earlier articles. Those hyperparameters were specific to those algorithm cases. The hyperparameters for NN can be much larger and only a tiny subset of them were explained here. Furthermore only two were experimented with to see their impact on NN performance.

Finally, the two datasets were used to train the MLP models and used for prediction. The fertility dataset was very simple and had a limited number of examples. It was used wholesale (without any training and test split) in the model and a single example was used to predict a binary classification outcome. The census dataset was more extensive and had a much larger quantity of examples. After the relevant features were enhanced or transformed to usable quantitative values (and other features removed), the data was split into train, validation and test datasets for training the model. This gave a way to examine metrics like accuracy and error rates and to do hyperparameter tuning to improve the performance across the datasets.

## References

[1] https://scikit-learn.org/stable/api/sklearn.neural_network.html

[2] https://www.geeksforgeeks.org/classification-using-sklearn-multi-layer-perceptron/

[3] https://garba.org/posts/2022/mlp/

[4] https://towardsdatascience.com/deep-neural-multilayer-perceptron-mlp-with-scikit-learn-2698e77155e/

[5] https://scikit-neuralnetwork.readthedocs.io/en/latest/module_mlp.html

[6] https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/neural_network/_multilayer_perceptron.py

[7] https://en.wikipedia.org/wiki/Multilayer_perceptron

[8] https://en.wikipedia.org/wiki/Perceptron

[9] https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

[10] https://en.wikipedia.org/wiki/File:Neuron3.png

[11] https://en.wikipedia.org/wiki/File:Colored_neural_network.svg

[12] Saleh, Hyatt. Machine Learning Fundamentals. Packt Publishing. 2018. Chapter 5. and source code https://github.com/TrainingByPackt/Machine-Learning-Fundamentals

[13] https://en.wikipedia.org/wiki/Stochastic_gradient_descent

[14] https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

[15] https://medium.com/data-science/stochastic-gradient-descent-explained-in-real-life-predicting-your-pizzas-cooking-time-b7639d5e6a32

[16] Census Income dataset: https://archive.ics.uci.edu/dataset/20/census+income

[17] R. Kohavi. "Census Income," UCI Machine Learning Repository, 1996. [Online]. Available: https://doi.org/10.24432/C5GP7S.

[18] Fertility dataset: https://archive.ics.uci.edu/ml/datasets/Fertility

[19] D. Gil and J. Girela. "Fertility," UCI Machine Learning Repository, 2012. [Online]. Available: https://doi.org/10.24432/C5Z01Z.

[20] https://dtmvamahs40ux.cloudfront.net/gl-academy/course/course-1281-Non-convex-optimization-We-utilize-stochastic-gradient-descent-to-find-a-local-optimum.jpg

## Appendix

This section covers some of the details of the error analysis discussed earlier in the theory section. It is meant as a refresher on some definitions and descriptions used for concepts appearing in the analysis and results sections.

In general, error analysis is used to compare different model performance. Metric selection is equally important because of their relevance to the type of problem being solved (i.e. regression or classification). The metric should be tied to the purpose of the study.

Accuracy, precision and recall are three scores that appear in classification reports. They can be used as metrics, but must be done after the trained model's predict method calls over the test datasets. Accuracy can be compared directly between models, while the precision and recall can be used to judge the quality of validation performance.

This will show if hyperparameter tuning is required and help check for overfitting. Lastly, these metrics are examined when the test dataset is used for determining the predictive performance on new data.

A few additional definitions and descriptions help clarify the rational behind doing this error analysis.

1. Bias can describe the condition of underfitting. Here the model does not generalize to the training set, so it will perform badly on unseen data. This can be fixed by changing ML algorithms or for ANN, going to a bigger network or training it for a longer time period.
2. Variance can describe the condition of overfitting. The model does not perform well to unseen data because it has overfit on the training dataset too tightly. This can be fixed by tuning the algorithm's hyperparameters. For ML algorithms like decision trees, they can be pruned to delete unneeded details. For ANN, regularization techniques can be added to reduce a neuron's influence on the overall result. More training data can be added as well to avoid overfitting.
3. Data mismatch happens when training and validation datasets do not follow the same distribution. So generalization on the training set does not translate to the validation set and hence eventually the test set. This can be fixed by making sure training and validation set follow the same distribution. This can be done by randomly shuffling together data from both sets to improve their distribution.
4. Overfitting the validation set is a similar problem of overfitting the training data. This could have happened by excessive optimization. This can be fixed by using the same techniques as variance above.

These descriptions provide some background and motivation of conditions that can occur when evaluating models. The metrics can be used for making comparisions. The descriptions identify problems that can occur as a result of the training process and ways to fix the conditions.