Model Performance And Fit - 1

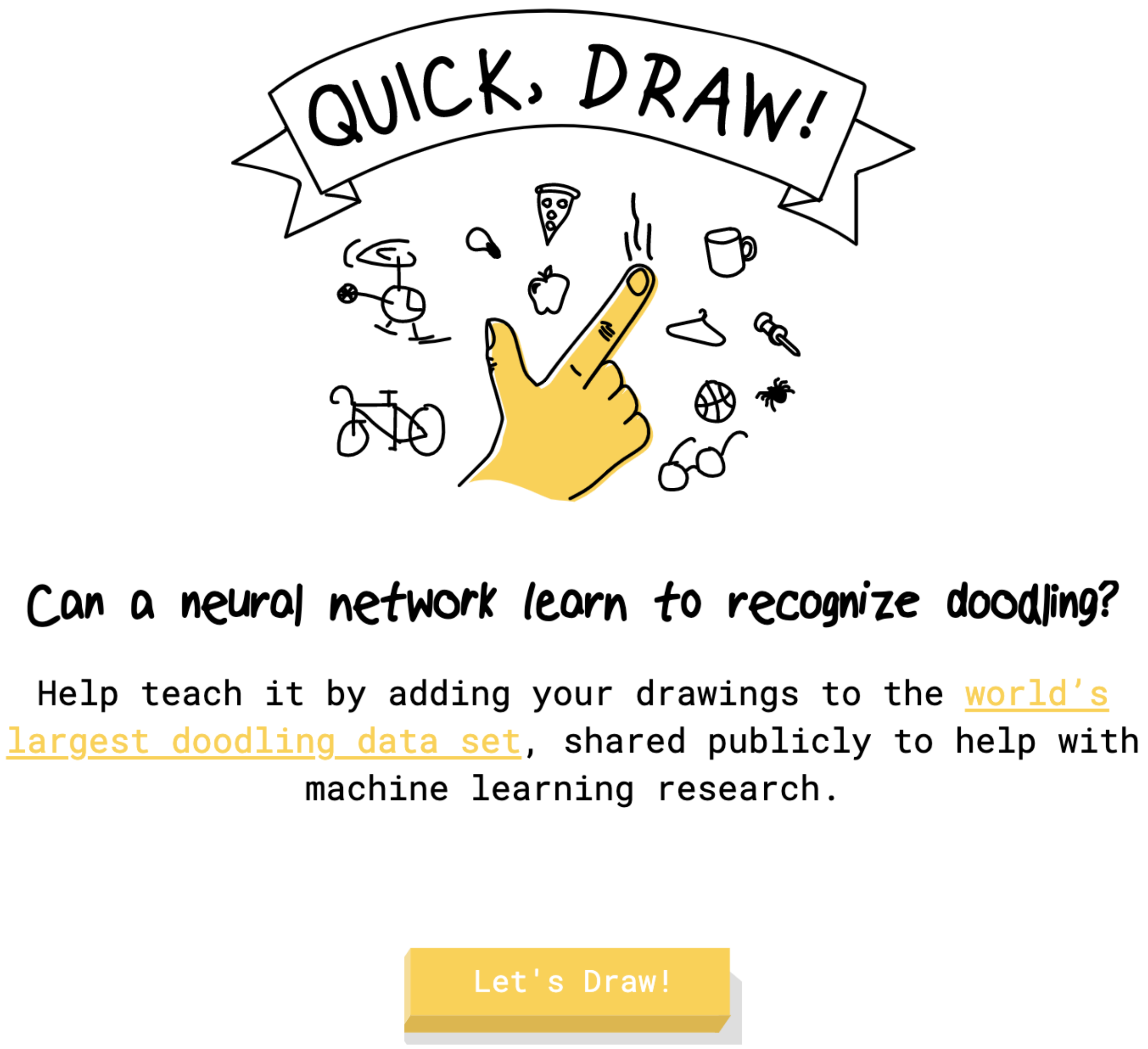
One should look for what is and not what he thinks should be. (Albert Einstein)

Model Performance and Fit: Topic introduction

In this part of the course, we will cover the following concepts:

- Implement a custom neural network to demonstrate model fit with different learning rates, epochs and batch sizes
- Understand loss functions and math behind gradient descent
- Assess and discuss methods to improve the fit of a neural network

Warm up: Quick, Draw!



- Quick, Draw! is an online game that uses neural network intelligence to guess what the drawings represent
- Visit the website to play ***Quick, Draw!***
- After completing, share how many of your doodles were correctly recognized

Module completion checklist

Objective	Complete
Summarize the role that batch size and epochs play in neural network training	
Implement a custom neural network to demonstrate model fit with different learning rates	

Epoch

- When the entire dataset is passed through a neural network, we consider one **epoch** of the training complete
  - Since the dataset can be too big to feed into the neural network, we divide it into **several smaller batches**
  - The number of batches needed to complete the epoch equals to the number of **iterations** per epoch the algorithm needs to make

Batch

- **Batch size** is a parameter we define while fitting the model
- It refers to the **number of observations our model takes at a time to make predictions** and update the weights in the following manner:
  - the algorithm **iterates** over one or more observations making predictions at the end of each batch
  - predictions are then compared and an error is calculated
  - this error term is then used to adjust the weights

2	NO	NO	2	11	11	81	17	331	654	
3	NO	NO	2	9	6.5	71.5	17	131	670	
4	NO	YES	1	1	4	44	18	120	1229	Batch=1
5	NO	YES	1	2	2.5	64	9	108	1454	
6	NO	YES	1	2.5	1	42.5	13	82	1518	
7	NO	YES	1	2	2	52	6	88	1518	
8	NO	YES	2	1	3	47.5	11	148	1362	
9	NO	NO	2	1	5	51	17	68	891	Batch=2
10	NO	NO	1	2	8.5	46	25	54	768	
11	NO	YES	1	2	6	50	15	41	1280	
										....
727	NO	YES	3	2.5	3	87	22	9	432	
728	NO	YES	2	3	1	60	21	247	1867	
729	NO	YES	2	3	1	60	9	644	2451	Batch=n
730	NO	NO	2	3	2	75	6	159	1182	
731	NO	NO	1	4	1	51	24	364	1432	

Batch order

- **Batches are selected randomly during each epoch**, unless we specify otherwise
  - Most problems are agnostic to the order of observations in batches and the order of batches
  - Some models, like time series problems, do need to have observations appear in order and need batch options set accordingly

5	NO	YES	1	1.5	2	70	12	15	416
10	NO	YES	1	2	6	50	15	41	1280
22	NO	YES	1	3	4	40	27	64	1466
26	NO	YES	3	11.5	11.5	94	7	69	1538
31	NO	YES	1	2	5	42	22	75	1468

Batch=1

33	NO	YES	2	10	8	93	19.5	81	1365
35	NO	YES	2	4	2	56	15	83	1844
36	NO	YES	1	2.5	7	49	13	86	1330
37	NO	YES	2	6.3	6.7	85.6	21.12	87	1009
40	NO	YES	1	2	8	42.5	24	89	2147

Batch=2

....

703	NO	NO	1	23	23	58	19	2512	5883
713	NO	NO	1	19.5	19.5	49	9	2622	4807
719	NO	NO	1	18	18	52	11	2795	4665
720	NO	NO	1	20	20	49	13	2795	5325
730	NO	NO	1	13.5	13.5	23.5	17	3252	3605

Batch=n

Batch size implications

- Batch size plays a big role in the time it takes to train the model and how well the model is doing
- With a **small** batch size, expect:
  - a longer train time
  - higher accuracy (which may lead to model overfitting)
  - it easily fits in computer memory
- With a **medium** batch size, expect:
  - a balance between training time and accuracy
  - better generalization to new data
  - it fits in memory
- With a **large** batch size, expect:
  - a faster training time, especially if training is done in parallel
  - it may generalize to new data poorly
  - it could be too big to fit into memory

Batch size implications (cont'd)

- A single batch can:
  - consist of a single observation (`batch_size = 1`), known as **stochastic gradient descent**
  - be as large as your dataset (`batch_size = num_observations`), known as **batch gradient descent**
  - be any value between `1` and the total `num_observations`, known as **mini-batch gradient descent**
- The default value for a batch size in TensorFlow and many other neural network frameworks is `32`

Batch size calculations

- `num_observations / batch_size` will provide the total number of resulting batches
- If the `batch_size` doesn't evenly divide into `num_observations`, the last batch will be smaller than the rest
    - If the model is sensitive to batches having the same outer dimension, set the `drop_remainder` argument to `True` to drop the smaller batch

Example: batch size, iterations, and epochs math

- Let's say you have a dataset with `1000` observations and you choose to set the batch size to `50` and the epochs to `200`:
  - `num_observations = 1000`
  - `batch_size = 50`
  - `epochs = 200`
- This means that the entire dataset will be split into `20` batches, each with `50` observations:
  - `num_batches = num_observations / batch_size = 20`
- This also means that `1` epoch will involve `20` batches or `20` updates in the model
  - `iterations = num_batches = 20`
- With `200` epochs, the model passes through the entire dataset `200` times, which means
  - `total_iterations = epochs*iterations = 4000`

Module completion checklist

Objective	Complete
Summarize the role that batch size and epochs play in neural network training	✔
Implement a custom neural network	

Loading packages

- Let's load the packages we will be using:

```
# Helper packages.
import os
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
import seaborn as sns

# Scikit-learn packages.
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import metrics
```

```
# TensorFlow and supporting packages.
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import load_model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

Directory settings

- Let's start by encoding the directory structure into `variables` in order to maximize the efficiency of your workflow
- Let the `data_dir` be the variable corresponding to your `data` folder

```
from pathlib import Path
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Load the data

- The `credit_card_data` dataset contains information about credit card **defaulters**
- Our goal is to predict if the customer will **default on a credit card payment**

```
credit_card = pd.read_csv(str(data_dir) + "/credit_card_data.csv")
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  PAY_AMT5  PAY_AMT6  default_payment_next_month
0   1      20000    2  ...         0         0                           1
1   2     120000    2  ...         0      2000                           1
2   3      90000    2  ...      1000      5000                           0
3   4      50000    2  ...      1069      1000                           0
4   5      50000    1  ...       689       679                           0

[5 rows x 25 columns]
```

Data preparation

- We need to make sure our data is in a suitable form to run through a neural network, which is why we must:
  - check the data for **NAs**
  - **transform the data to numeric values** *(if it's categorical, make sure the data is encoded)*
  - split data into train, test, and validation
  - normalize data
  - examine the target variable imbalance

Remember that the order of operations matters!

- Ideally, all data transformations should happen after the data has been split
- In this instance, we will check for NAs and encode categorical variables before the split since this will not significantly affect the results and will keep our code more concise

Data prep: convenience function

- We wrote a time-saving function to perform all of the cleaning and split steps on the credit card dataset at once!

```python
def data_prep(df):

    # Fill missing values with mean
    df = df.fillna(df.mean()['BILL_AMT1'])
    # Drop an unnecessary identifier column.
    df = df.drop('ID',axis = 1)

    # Convert 'sex' into dummy variables.
    sex = pd.get_dummies(df['SEX'], prefix = 'sex', drop_first = True)
    # Convert 'education' into dummy variables.
    education = pd.get_dummies(df['EDUCATION'], prefix = 'education', drop_first = True)
    # Convert 'marriage' into dummy variables.
    marriage = pd.get_dummies(df['MARRIAGE'], prefix = 'marriage', drop_first = True)
    # Drop `sex`, `education`, `marriage` from the data.
    df.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)

    # Concatenate `sex`, `education`, `marriage` dummies to our dataset.
    df = pd.concat([df, sex, education, marriage], axis=1)
```

```python
    # Separate predictors from data.
    X = df.drop(['default_payment_next_month'], axis=1)

    # Separate target from data.
    y = df['default_payment_next_month']
```

Data prep: convenience function (cont'd)

```
    # Set the seed to 1.
    np.random.seed(1)

    # Split data into train, test, and validation set, use a 70 - 15 - 15 split.
    # First split data into train-test with 70% for train and 30% for test.
    X_train, X_test, y_train, y_test = train_test_split(X.values,
                                                        y,
                                                        test_size = .3,
                                                        random_state = 1)


    # Then split the test data into two halves: test and validation.
    X_test, X_val, y_test, y_val = train_test_split(X_test,
                                                    y_test,
                                                    test_size = .5,
                                                    random_state = 1)


    print("Train shape:", X_train.shape, "Test shape:", X_test.shape, "Val shape:", X_val.shape)
```

```
    # Transforms features by scaling each feature to a given range.
    # The default is the range between 0 and 1.
    min_max_scaler = preprocessing.MinMaxScaler()
    X_train_scaled = min_max_scaler.fit_transform(X_train)
    X_test_scaled = min_max_scaler.transform(X_test)
    X_val_scaled = min_max_scaler.transform(X_val)

    return X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val
```

Data prep

```
X_train_scaled, X_test_scaled, X_val_scaled, y_train, y_test, y_val = data_prep(credit_card)
```

```
Train shape: (21000, 30) Test shape: (4500, 30) Val shape: (4500, 30)
```

Define and compile a sequential model

- Let's create a convenience function to define and compile the model with an input layer, two hidden layers, and an output layer

```
def create_model(lr = .01):
  # Let's set the seed so that we can reproduce the results.
  tf.random.set_seed(1)
  opt = Adam(learning_rate = lr) # <- set optimizer

  model = Sequential([
          Dense(32, activation='relu', input_dim=30),#<- set input and 1st hidden layer
          Dense(32, activation='relu'),              #<- set 2nd hidden layer
          Dense(1, activation='sigmoid')             #<- set output layer

  ])

  model.compile(optimizer = opt,                 #<- set optimizer
                loss='binary_crossentropy',      #<- set loss function to binary_crossentropy
                metrics=['accuracy'])            #<- set performance metric
  return model
```

- We will create models with different numbers of learning rates to compare how those parameters affect loss and accuracy
- We will use the default `batch_size - 32` in this experiment

Default learning rate

- We have set the learning rate to default to `0.01` in the `create_model()` function, so we don't have to specify it explicitly

```
lr_default = create_model().fit(X_train_scaled, y_train,
                               epochs = 25,
                               validation_data=(X_val_scaled,y_val))
```

```
Epoch 1/25
657/657 [==============================] - 1s 1ms/step - loss: 0.4889 - accuracy: 0.7938 -
val_loss: 0.4522 - val_accuracy: 0.8196
Epoch 2/25
657/657 [==============================] - 1s 942us/step - loss: 0.4496 - accuracy: 0.8161 -
val_loss: 0.4504 - val_accuracy: 0.8162
...
Epoch 24/25
657/657 [==============================] - 1s 852us/step - loss: 0.4372 - accuracy: 0.8169 -
val_loss: 0.4337 - val_accuracy: 0.8202
Epoch 25/25
657/657 [==============================] - 1s 827us/step - loss: 0.4294 - accuracy: 0.8236 -
val_loss: 0.4355 - val_accuracy: 0.8216
```

High learning rate

- Let's set the learning rate to a very high number like `0.75`

```
# Set learning rate to 0.75.
lr_high = create_model(lr = .75).fit(X_train_scaled, y_train,
                                      epochs = 25,
                                      validation_data=(X_val_scaled, y_val))
```

```
Epoch 1/25
657/657 [==============================] - 1s 930us/step - loss: 1.5291 - accuracy: 0.7780 -
val_loss: 0.5284 - val_accuracy: 0.7798
Epoch 2/25
657/657 [==============================] - 1s 831us/step - loss: 0.5381 - accuracy: 0.7773 -
val_loss: 0.5272 - val_accuracy: 0.7798
...
Epoch 24/25
657/657 [==============================] - 1s 840us/step - loss: 0.5459 - accuracy: 0.7751 -
val_loss: 0.5336 - val_accuracy: 0.7798
Epoch 25/25
657/657 [==============================] - 1s 859us/step - loss: 0.5362 - accuracy: 0.7824 -
val_loss: 0.5274 - val_accuracy: 0.7798
```

Low learning rate

- Let's set the learning rate to a very low number like `0.0001`
  - Let's also increase the number of epochs here to `50` since the learning rate is low

```
lr_low = create_model(lr=.0001).fit(X_train_scaled, y_train,
                                    epochs = 50, #<- increase the number of epochs
                                    validation_data=(X_val_scaled, y_val))
```

```
Epoch 1/50
657/657 [==============================] - 1s 952us/step - loss: 0.5974 - accuracy: 0.7577 -
val_loss: 0.5131 - val_accuracy: 0.7798
Epoch 2/50
657/657 [==============================] - 1s 841us/step - loss: 0.5114 - accuracy: 0.7773 -
val_loss: 0.4983 - val_accuracy: 0.7800
...
Epoch 49/50
657/657 [==============================] - 1s 832us/step - loss: 0.4391 - accuracy: 0.8223 -
val_loss: 0.4392 - val_accuracy: 0.8227
Epoch 50/50
657/657 [==============================] - 1s 830us/step - loss: 0.4328 - accuracy: 0.8254 -
val_loss: 0.4393 - val_accuracy: 0.8233
```

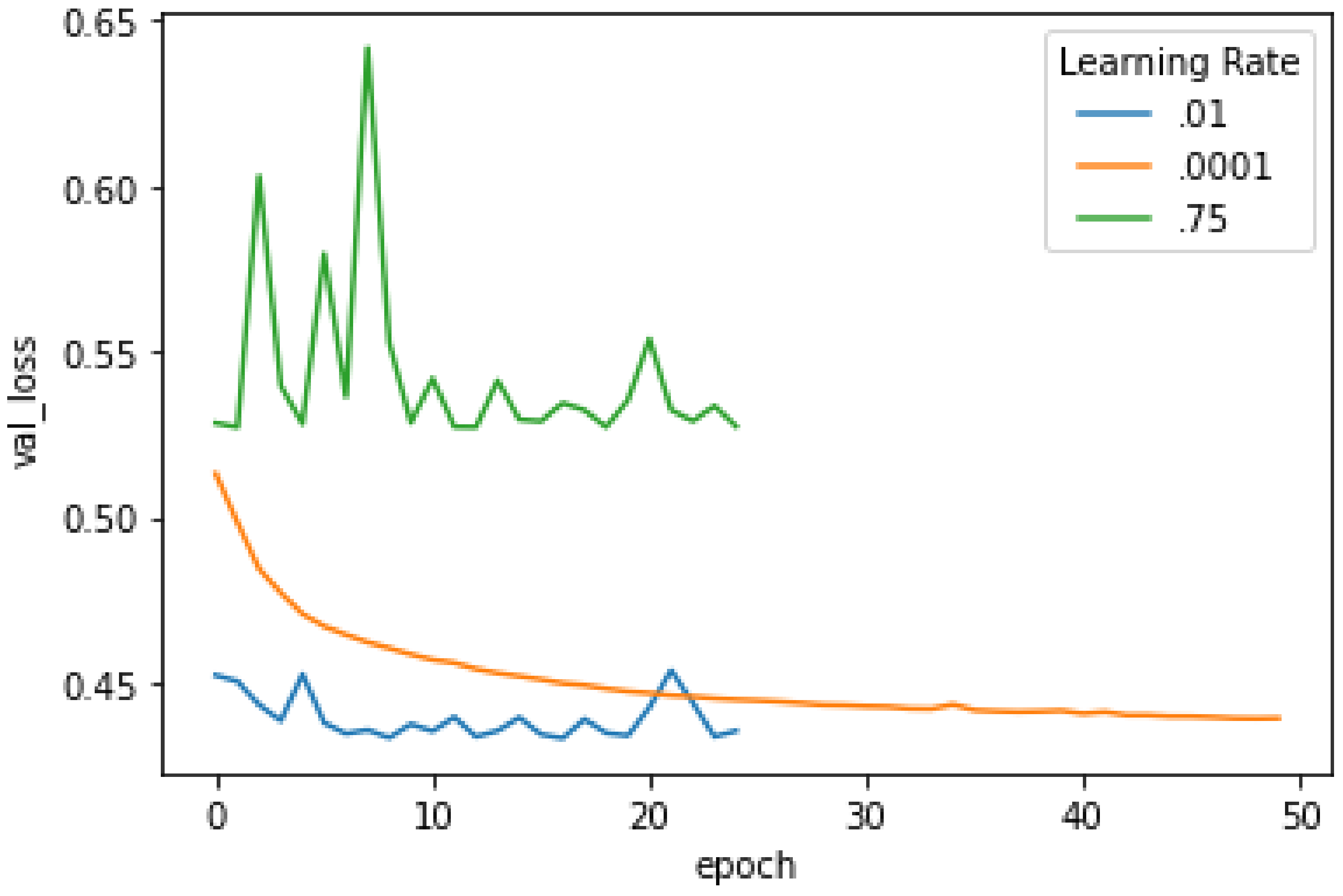
Visualize results for learning rates

- Let's create a dataframe with the loss and accuracy for training and validation data along with their corresponding epochs and learning rates

```
learn_rates = []

for exp, result in zip([lr_default, lr_low, lr_high], [".01", ".0001", ".75"]):

  df = pd.DataFrame.from_dict(exp.history)
  df['epoch'] = df.index.values
  df['Learning Rate'] = result

  learn_rates.append(df)

df_learning = pd.concat(learn_rates)
df_learning['Learning Rate'] = df_learning['Learning Rate'].astype('str')
```

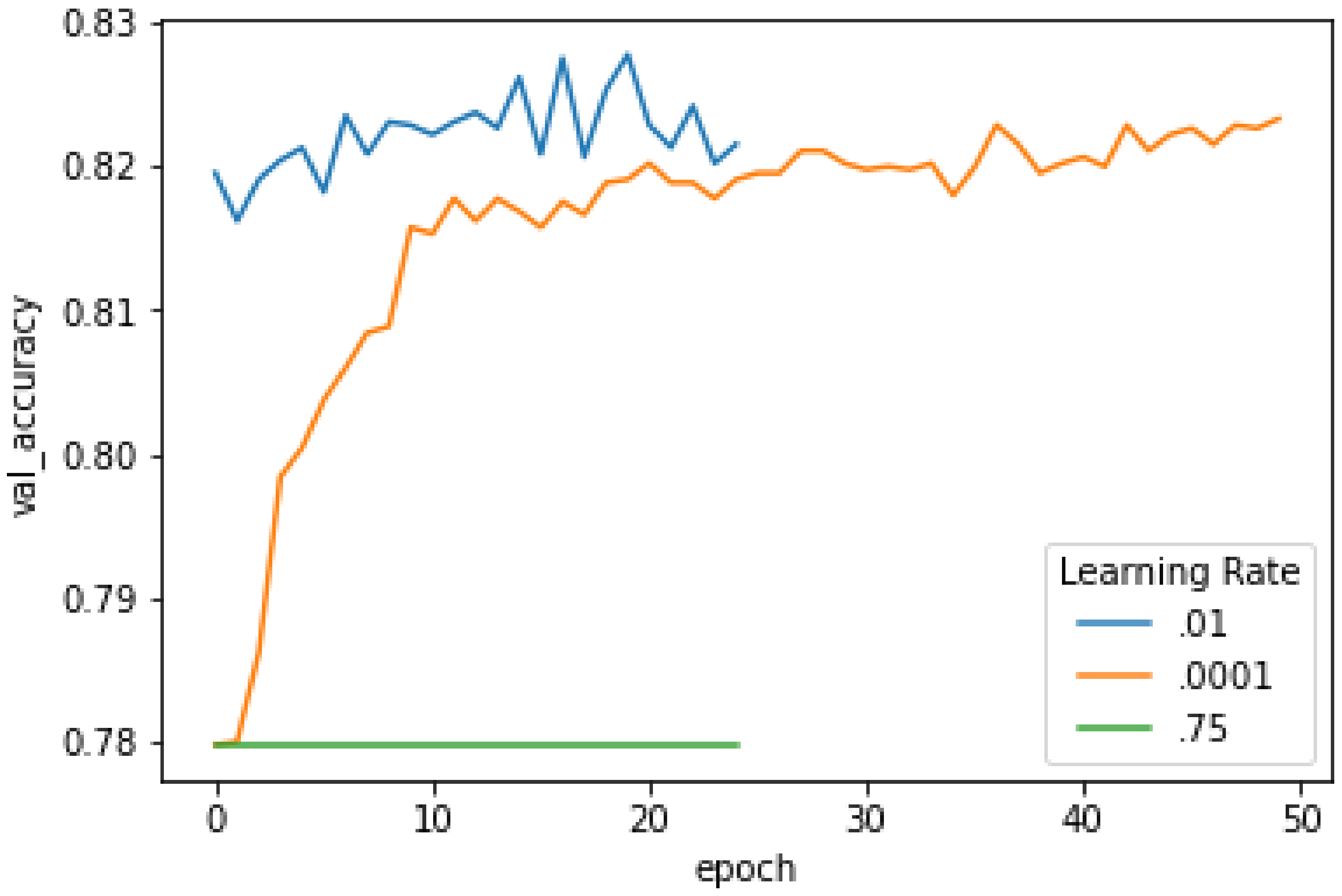
Visualize results for learning rates (cont'd)

```
sns.lineplot(x='epoch', y='val_loss', hue='Learning Rate', data=df_learning)
```



Visualize results for learning rates (cont'd)

```
sns.lineplot(x='epoch', y='val_accuracy', hue='Learning Rate', data=df_learning)
```



- We obtain the best results when the learning rate is set to 0.01 and the model seems to clearly underfit when the learning rate is high

Knowledge check

Module completion checklist

Objective	Complete
Summarize the role that batch size and epochs play in neural network training	✔
Implement a custom neural network to demonstrate model fit with different learning rates	✔

Congratulations on completing this module!

You are now ready to try tasks 1-7 in the Exercise for this topic