In [ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

In [ ]:
```
# Please do not change this cell because some hidden tests might depend on i
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls tests >/dev/null 2>&1
if [ ! $? = 0 ]; then
 # Check if the repository exists before trying to clone it
 if git ls-remote https://github.com/cs187-2025/lab1-5.git >/dev/null 2>&1;
   rm -rf .tmp
   git clone https://github.com/cs187-2025/lab1-5.git .tmp
   mv .tmp/tests ./
   if [ -f .tmp/requirements.txt ]; then
     mv .tmp/requirements.txt ./
   fi
   rm -rf .tmp
 else
   echo \"Repository https://github.com/cs187-2025/lab1-5.git does not exist
 fi
fi
if [ -f requirements.txt ]; then
  python -m pip install -q -r requirements.txt
fi
""")
```

%%latex \newcommand{\vect}[1]{\mathbf{#1}} \newcommand{\cnt}[1]{\sharp(#1)} \newcommand{\argmax}[1]{\underset{#1}{\operatorname{argmax}}} \newcommand{\softmax}{\operatorname{softmax}} \newcommand{\Prob}{\Pr} \newcommand{\given}{\,|\,}

As you've seen, a typical pipeline for NLP applications based on supervised machine learning involves several standard components:

1. Loading of annotated textual corpora.

2. Tokenization and normalization of the text.
3. Distributing instances into subcorpora, for instance, training, development, and test corpora.
4. Training of models on training data, using development data for model selection.
5. Evaluation of the models on test data.

Rather than recapitulate all of these component tasks for each application, standard packages have been developed to facilitate them. In order to facilitate your own experimentation, it's time to make use of some of these packages to scale up your ability to build and test models. That is the subject of this lab.

Huggingface is a supplier of open tools for a variety of machine learning applications. Huggingface's `datasets` and `tokenizers` libraries provide a uniform system for establishing dataset objects that contain multiple examples, each of which may have multiple features. These libraries provide functions to preprocess, tokenize, or postprocess the data. Dataset objects can be easily split into parts (training and test, for instance), or turned into a sequence of small batches for processing by models. Most importantly, as you will see in future labs, these data loading libraries can be easily integrated with Huggingface's `transformers` library which is widely used in NLP applications these days.

This lab provides an introduction to using `datasets`, `tokenizers`, and PyTorch in preparation for its appearance in later labs and homework problem sets.

After this lab, you should be able to

- Read data loading code implemented using `datasets` and `tokenizers` and understand what it is intending to accomplish.
- Run experiments training and testing simple feed-forward neural networks using PyTorch.

New bits of Python used for the first time in the *distribution version* of this lab include:

- Datasets
  - `datasets.load_dataset`
  - `datasets.Dataset`
  - `datasets.Dataset.shuffle`
  - `datasets.Dataset.train_test_split`
  - `datasets.Dataset.add_column`
  - `datasets.Dataset.map`

    - `datasets.Dataset.class_encode_column`
    - `datasets.Dataset.align_labels_with_mapping`
- Tokenizers
    - `tokenizers.Tokenizer`
    - `tokenizer.Tokenizer.encode`
    - `tokenizer.Tokenizer.decode`
    - `tokenizer.Tokenizer.get_vocab`
    - `tokenizer.Tokenizer.train_from_iterator`
    - `tokenizer.processors.TemplateProcessing`
- Transformers
    - `transformers.PreTrainedTokenizerFast`
- Torch
    - `torch.utils.data.DataLoader`
    - `torch.nn.Module.eval`
    - `torch.nn.Linear`
    - `torch.nn.Sigmoid`
    - `torch.distributions.normal.Normal`
    - `torch.distributions.uniform.Uniform`
- `tqdm.tqdm` (for generating progress bars)

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `torch.Tensor.backward`
- `torch.optim.Optimizer.step`

Preparation – Loading packages and data {-}

In [ ]:
```
import copy
import math
import random
import matplotlib.pyplot as plt
import os
import re
import sys
import warnings
import wget
import csv

import torch
import torch.distributions as ds
import torch.nn as nn
import torch.nn.functional as F
```

```
import datasets

from copy import deepcopy

from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.pre_tokenizers import Whitespace
from tokenizers.processors import TemplateProcessing
from tokenizers import normalizers
from tokenizers.models import WordLevel
from tokenizers.trainers import WordLevelTrainer

from transformers import PreTrainedTokenizerFast

from torch import optim
from tqdm.notebook import tqdm
```

In [ ]:
```
# Set up plotting
plt.style.use('tableau-colorblind10')

# Fix random seed for replicability
random_seed = 1234
random.seed(random_seed)
torch.manual_seed(random_seed)

## GPU check
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)
```

Manipulating text corpora with `datasets`

You'll use Huggingface's `datasets` to load the *Green Eggs and Ham* (GEaH) dataset.

We start with reading in the data and performing some ad hoc cleaning (removing comment lines and blank lines).

In [ ]:
```
def strip_file(text):
    """strip #comments and empty lines from `text` string"""
    result = ""
    for line in text.split("\n"):
        line = line.strip()              # trim whitespace
        line = re.sub('#.*$', '', line)  # trim comments
        if line != '':                   # drop blank lines
            result += line + '\n'
    return result

# Read the GEaH data and write out a corresponding CSV file
os.makedirs('data', exist_ok=True)
wget.download("https://github.com/nlp-course/data/raw/master/Seuss/seuss - 1
with open('data/seuss - 1960 - green eggs and ham.txt', 'r') as fin:
    with open('data/geah.csv', 'w') as fout:
        writer = csv.writer(fout)
```

```
        writer.writerow(('label','text'))
        text = strip_file(fin.read())
        for line in text.strip().split('\n'):
            label, text = line.split('\t')
            writer.writerow((label, text))
```

Constructing training and test datasets

Take a look at the file `geah.csv`, which we've just processed and placed into the sibling `data` folder.

In [ ]:
```
shell('head "data/geah.csv"')
```

Notice the structure of this corpus. Each line contains a sentence from the book, preceded by a label that provides the speaker of that sentence. The speaker and sentence are separated by a comma. The data is thus set up properly for using Huggingface's `datasets.load_dataset` using its `"CSV"` (comma-separated values) format.

Now, you can set up the dataset using `load_dataset`. It should look for the CSV data in the file `data/geah.csv`, and should pass the argument `split='all'` such that a `dataset.Dataset` object instead of `dataset.DatasetDict` is returned. We'll call the dataset `geah`.

In [ ]:
```
#TODO: Set up the dataset using `load_dataset`. You should pass
# `split='all'` such that a `dataset.Dataset` object is returned.
geah = ...
```

In [ ]:
```
grader.check("dataset_setup")
```

We can see that the loaded dataset has two features: label and text.

In [ ]:
```
geah
```

We can access examples from this dataset simply by indexing it similar to how we index a Python list.

In [ ]:
```
# The first example
geah[0]
```

In [ ]:
```
# The second to the fifth examples
geah[1:5]
```

All `dataset.Dataset` objects have a `train_test_split` method that splits the dataset into two pieces, for instance, to have a separate training and test set. Use the `train_test_split` method to generate a 70%/30% split of the GEaH corpus into two

subsets called `train` and `test`. You should pass `seed=random_seed` to make sure the splitting is deterministic and to pass the unit tests.

In [ ]:
```
#TODO: Split geah into 70% training data and 30% test data
# Make sure to pass `random_seed` as the seed.
train_test = \
    ...
train = train_test['train']
test = train_test['test']
```

In [ ]:
```
grader.check("dataset_split")
```

For the label feature, we can convert it to integer label ids to facilitate processing it with models such as neural networks using `datasets.Dataset.class_encode_column`.

In [ ]:
```
train = train.add_column('label_id', train['label'])
test = test.add_column('label_id', test['label'])
train = train.class_encode_column('label_id')

label2id = train.features['label_id']._str2int
test = test.class_encode_column('label_id')
test = test.align_labels_with_mapping(label2id, "label_id")
```

The new feature `label_id` stores the label ids.

In [ ]:
```
train[1:5]
```

Tokenization

Now let's turn our attention to text features. As we have explored in lab 1-1, text can be represented as a sequence of integer word ids. In Huggingface datasets, we need to create a tokenizer which both specifies how to tokenize text into a sequence of tokens and also internally maintains a *vocabulary* which establishes the mapping from types to indices. We will use the training corpus to establish the vocabulary.

In [ ]:
```
unk_token = '[UNK]' # a token representing unknown (out-of-vocabulary) token
pad_token = '[PAD]' # a padding token

# Instantiate a tokenizer
tokenizer = Tokenizer(WordLevel(unk_token=unk_token))  # WordLevel is a simp
                                                       # that doesn't split
tokenizer.pre_tokenizer = Whitespace()                 # pre-tokenizer split
tokenizer.normalizer = normalizers.Lowercase()         # normalizer lowercas

# Build the vocabulary from training data
trainer = WordLevelTrainer(special_tokens=[pad_token, unk_token]) # trainer
                                                                  # building
tokenizer.train_from_iterator(train['text'], trainer=trainer)
```

Let's try out the tokenizer on some examples.

In [ ]:
```
encoded = tokenizer.encode('I am Sam.')
print (f'tokens: {encoded.tokens}')
print (f'word ids: {encoded.ids}')
print (f'converted back: {tokenizer.decode(encoded.ids, skip_special_tokens=
```

In [ ]:
```
encoded = tokenizer.encode('An example with many unknown tokens.')
print (f'tokens:         {encoded.tokens}')
print (f'word ids:       {encoded.ids}')
print (f'converted back: {tokenizer.decode(encoded.ids, skip_special_tokens=
```

A digression: Postprocessing

In future labs we will also use `tokenizer.post_processor` which can be used to post-process the tokenized text. In the following example we will add exclamation points `!` in the beginning and end of the tokenized text using TemplateProcessing.

In [ ]:
```
exclamation_point = '!'
post_processing_tokenizer = deepcopy(tokenizer) # we clone the tokenizer for
post_processing_tokenizer.post_processor = \
  TemplateProcessing(single=f"{exclamation_point} $A {exclamation_point}",
                     special_tokens=[(exclamation_point,
                                      post_processing_tokenizer.token_to_id(

encoded = post_processing_tokenizer.encode('I am Sam.')
print (f'tokens: {encoded.tokens}')
print (f'word ids: {encoded.ids}')
print (f'converted back: {post_processing_tokenizer.decode(encoded.ids, skip
```

With a tokenizer, we can convert text strings into integer word ids using `datasets.Dataset.map`. But first we need to wrap `tokenizer` with the `transformers.PreTrainedTokenizerFast` class. `transformers.PreTrainedTokenizerFast` provide a convenient interface for tokenizers that are compatible with PyTorch tensors.

In [ ]:
```
hf_tokenizer = PreTrainedTokenizerFast(tokenizer_object=tokenizer,
                                       pad_token=pad_token,
                                       unk_token=unk_token)
```

This `hf_tokenizer` takes in a text string and converts it to a sequence of word ids stored in `input_ids` of the returned dictionary. (You can ignore `token_type_ids` and `attention_mask` for now.)

In [ ]:
```
text = 'I am Sam.'
hf_tokenizer(text)
```

We can use the `hf_tokenizer` to get a list of tokens.

In [ ]:
```
hf_tokenizer.tokenize(text)
```

`hf_tokenizer` can also be used to decode some encoded text.

In [ ]:
```
encoded_text = hf_tokenizer(text).input_ids
decoded_text = hf_tokenizer.decode(encoded_text)
decoded_tokens = hf_tokenizer.convert_ids_to_tokens(encoded_text)
print(f"text: {text}")
print(f"encoded text: {encoded_text}")
print(f"decoded text: {decoded_text}")
print(f"decoded token list: {decoded_tokens}")
```

Note that `hf_tokenizer` also stores the vocabulary and the special tokens we provided it in the initialization.

In [ ]:
```
print(f"hf_tokenizer.unk_token: {hf_tokenizer.unk_token}")
print(f"hf_tokenizer.unk_token_id: {hf_tokenizer.unk_token_id}")
print(f"hf_tokenizer.pad_token: {hf_tokenizer.pad_token}")
print(f"hf_tokenizer.pad_token_id: {hf_tokenizer.pad_token_id}")
print(f"hf_tokenizer.get_vocab():\n {hf_tokenizer.get_vocab()}")
print(f"len(hf_tokenizer) == len(hf_tokenizer.get_vocab()): {len(hf_tokenize
```

Now we can convert the training and test sets into word ids using `datasets.Dataset.map`.

In [ ]:
```
def encode(example):
    return hf_tokenizer(example['text'])

train = train.map(encode)
test = test.map(encode)
```

Let's take a look at one example from the mapped dataset. The new feature `input_ids` contain the word ids.

In [ ]:
```
train[0]
```

Again, we can view several examples at once.

In [ ]:
```
train[1:5]
```

The text and label features now have vocabularies that maps the elements of the vocabulary to integer index representations of the elements, accessible via the below code.

In [ ]:
```
text_vocab = tokenizer.get_vocab()
label_vocab = train.features['label_id']._str2int
```

In [ ]:
```
label_vocab
```

In [ ]:
```
text_vocab
```

How many elements are there in these vocabularies? You can use the `len` function to find out.

In [ ]:
```
#TODO: Calculate the sizes of label_vocab and text_vocab
label_vocab_size = ...
text_vocab_size = ...
```

In [ ]:
```
grader.check("vocab_sizes")
```

In [ ]:
```
print(f"label vocabulary size is {label_vocab_size}\n"
      f"text vocabulary size is {text_vocab_size}")
```

Operations over datasets

We now have training and test datasets. You can experiment with the kinds of operations you'll need to do to implement models like Naive Bayes or logistic regression.

For instance, you can inspect an example from the dataset.

In [ ]:
```
example = train[1] # the second instance
print (f"text: {example['text']}\n"
       f"label: {example['label']}")
```

You might also need to iterate over the different class labels (`label_vocab`) or the word types (`text_vocab`). Define a function that iterates over a vocabulary and prints each one out in alphabetical order along with their corresponding ids like this:

```
>>> print_vocab(label_vocab)
GUY: 0
SAM: 1
```

In [ ]:
```
#TODO: print out vocabulary in alphabetical order with ids
def print_vocab(vocab):
    ...
```

In [ ]:
```
grader.check("print_vocab")
```

We can use the `print_vocab` function to print out the different class labels.

In [ ]:
```
print_vocab(label_vocab)
```

Similarly, we can also print out `text_vocab` .

In [ ]:
```
print_vocab(text_vocab)
```

Other simple calculations that will be useful in implementing the various models:

1. Counting how many instances there are in a dataset.

2. Counting how many instances of a certain class there are in a dataset.
3. Counting how many tokens of a certain type there are in the text of an instance.

Let's write functions for these. They'll come in handy in the first problem set.

In [ ]:
```
#TODO - 1. Counting how many instances there are in a dataset.
def count_instances(dataset):
    ...
```

In [ ]:
```
grader.check("count_instances")
```

In [ ]:
```
#TODO - 2. Counting how many instances of a certain class there are in a dat
# Note: recall that you can access the label of an instance using `instance[
def count_instances_class(dataset, label):
    ...
```

In [ ]:
```
grader.check("count_instances_class")
```

In [ ]:
```
#TODO - 3. Counting how many tokens of a certain type there are in the text
# Hint: we have provided code for extracting tokens from the text of an inst
def count_tokens_instance(instance, tokentype):
    tokens = tokenizer.encode(instance['text']).tokens
    ...
```

In [ ]:
```
grader.check("count_tokens_instances")
```

Recall that the purpose of these tokenizer objects is to map back and forth between strings and word ids. Below provides an example of how to do that.

In [ ]:
```
example = train[1]
print (f'text: {example["text"]}')
word_ids = example['input_ids']
print (f"Mapped to word ids: {word_ids}\n"
       f"Mapped back: {hf_tokenizer.decode(word_ids)}")
```

We can also do the conversion for a batch of examples, which will be very useful in future labs when we take advantage of the parallelism provided by parallel hardware such as GPUs.

In [ ]:
```
examples = train[1:5]
print (f'text: {examples["text"]}')
word_ids_batch = examples['input_ids']
print (f"Mapped to word ids: {word_ids_batch}\n"
       f"Mapped back: {hf_tokenizer.batch_decode(word_ids_batch)}\n"
       f"Mapped back skipping padding: {hf_tokenizer.batch_decode(word_ids_b
```

You might have noticed that in the above example, different sentences are of different lengths. This makes it hard for processing on parallel hardware, and we need to *pad* all sentences to the length of the longest sentence in each batch before converting them to tensors, as shown below. (You will see the below code in project 1 and in future labs.)

In [ ]:
```
pad_id = hf_tokenizer.pad_token_id
BATCH_SIZE = 32

# Defines how to batch a list of examples together
def collate_fn(examples):
    batch = {}
    bsz = len(examples)
    label_ids = []
    for example in examples:
        label_ids.append(example['label_id'])
    label_batch = torch.LongTensor(label_ids).to(device)
    input_ids = []
    for example in examples:
        input_ids.append(example['input_ids'])
    max_length = max([len(word_ids) for word_ids in input_ids])
    text_batch = torch.zeros(bsz, max_length).long().fill_(pad_id).to(device
    for b in range(bsz):
        text_batch[b][:len(input_ids[b])] = torch.LongTensor(input_ids[b]).t

    batch['label_ids'] = label_batch
    batch['input_ids'] = text_batch
    return batch

train_iter = torch.utils.data.DataLoader(train, batch_size=BATCH_SIZE, colla
test_iter = torch.utils.data.DataLoader(test, batch_size=BATCH_SIZE, collate
```

Let's look at a single batch from one of these iterators. Each sentence is the same length, with padding as needed with the `PAD` token, which has id `0`.

In [ ]:
```
next(iter(train_iter))
```

Training and testing with PyTorch

Past labs have shown that all of the detail about

- establishing models and their parameters,
- using them to calculate the outputs for some inputs,
- training them to optimize the parameters via stochastic gradient descent, and
- evaluating them by testing on held-out data

is tedious to manage. Fortunately, it is also so formulaic, at least for a certain class of models, that general tools can be deployed to manage the process. In the remainder of this lab, you'll use one such tool, PyTorch. For simplicity, rather than a natural-language

task, you'll be training a model to fit a curve; it has an especially simple structure: one scalar input and one scalar output.

Generating training and test data

We start by generating some training and test data. The data is generated as a noisy sine function, calculated by the function `sinusoid`. (Here we make use of the PyTorch distributions package, which was imported above as `ds`.)

In [ ]:
```
def sinusoid(x, amplitude=1., phase=0., frequency=1., noise=1e-5):
    """Returns the values on input(s) `x` of a sinusoid determined by `ampli
       `phase`, and angular `frequency`, with some added normal noise with v
       given by `noise`."""
    normal_noise = ds.normal.Normal(torch.tensor([0.0]), torch.tensor([noise
    noise_sample = normal_noise.sample(x.size()).view(-1)
    y = amplitude * torch.sin(x * frequency + phase) + noise_sample
    return y
```

We can generate data for training and testing by sampling this function.

In [ ]:
```
def sample_input(func, count, bound, **kwargs):
    """Returns `count` samples of x-y pairs of function `func`, with the x
       values sampled uniformly between +/-`bound`. The `kwargs` are passed
       on to `func`."""
    input_unif = ds.uniform.Uniform(-bound, +bound)
    x = input_unif.sample(torch.Size([count]))
    y = func(x, **kwargs)
    return x, y
```

To give a sense of what a data sample looks like, we plot a sample of 100 points.

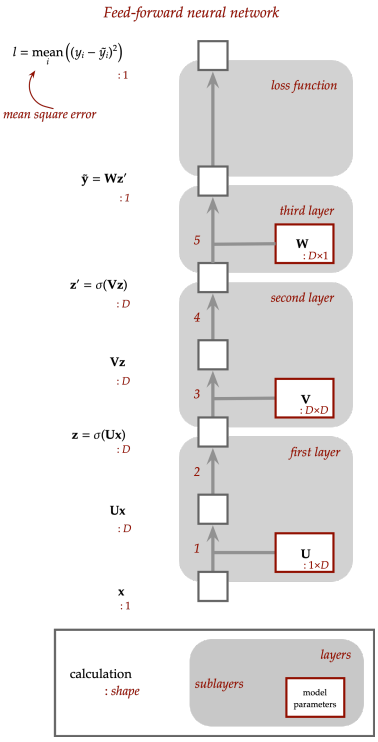
In [ ]:
```
def plot_sample(data):
    """Plots `data` given as a single pair of inputs and outputs."""
    X, Y = data
    plt.plot(X.tolist(), Y.tolist(), '.')
    plt.xlabel('Input')
    plt.ylabel('Output')
    # we cannot use plt.show() because otter-grader does not support it
```

In [ ]:
```
plot_sample(sample_input(sinusoid, 100, 5, noise=0.1))
```

Specifying a feed-forward neural network

The model that we will train to predict the output of this function based on a sample will consist of a series of sublayers as depicted in the figure at right. At the bottom of the figure, we start with $\mathbf{x}$, the scalar input (of dimensionality 1 as shown in the "shape" designation). The first layer is a perceptron layer, composed of a linear sublayer (with weights $\mathbf{U}$) followed by a sigmoid sublayer. Since $\mathbf{U}$ is of dimensionality $1 \times D$, the

output is a vector of dimensionality D. (We refer to D as the *hidden dimension*.) Then comes another perceptron layer with input and output each of dimensionality D. Finally, a single linear layer reduces the dimensionality back to the predicted scalar output $\tilde{y}$ of dimensionality 1. The loss is calculated as the mean square error of $\tilde{y}$ relative to y. (In this case, taking the mean for a single example is irrelevant, since y is a scalar, though when training in "batches", the mean would be taken over the batch.)



We define a class `FFNN` (**f**eed-**f**orward **n**eural **n**etwork), which inherits from the `nn.Module` class, PyTorch's class for neural network models. It takes an argument `hidden_dim` which is the size of the hidden layers, D in the figure.

The parameters of this model – the values that will be adjusted to minimize the loss – are the elements of the tensors $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$. They don't appear explicitly in the code, but are PyTorch *parameters* created and tracked when the corresponding sublayers are created using `nn.Linear`. That's the wonder of using PyTorch – so much happens under the hood. But we can get access to the parameters because the `nn.Module` class provides a `parameters` method that returns an iterator over the parameters. (We use it to initialize the parameters to random values.)

In [ ]:
```
class FFNN(nn.Module):
    def __init__(self, hidden_dim, init_low=-2, init_high=2):
        super().__init__()
        # dimensionality of hidden layers
        self.hidden_dim = hidden_dim
        # establishing the sublayers -- two perceptrons (each with a linear
        # layer and a sigmoid) and a final linear layer -- and their paramet
        self.sublayer1 = nn.Linear(1, hidden_dim)          # U: 1 X D
        self.sublayer2 = nn.Sigmoid()
        self.sublayer3 = nn.Linear(hidden_dim, hidden_dim) # V: D X D
        self.sublayer4 = nn.Sigmoid()
        self.sublayer5 = nn.Linear(hidden_dim, 1)          # W: D X 1

        # initialize parameters randomly
        torch.manual_seed(random_seed)
        for p in self.parameters():
            p.data.uniform_(init_low, init_high)
        # save a copy of the parameters to allow resetting
        self.init_state = copy.deepcopy(self.state_dict())

    # Resetting state: If you want to rerun a model, say, with a different
    # training regime, you can reset the model's parameter state using
    #    model.reset_state()
    # before retraining, e.g.,
```

```
    #    train_model(model, criterion, optim, train_data, n_epochs=50)
    def reset_state(self):
        self.load_state_dict(self.init_state)

    def forward(self, x):
        # first perceptron layer
        z = self.sublayer2(self.sublayer1(x))
        # second perceptron layer
        z_prime = self.sublayer4(self.sublayer3(z))
        # final linear layer
        return self.sublayer5(z_prime)
```

We can build a model by instantiating the `FFNN` class. We'll do so with a hidden dimension of 4, being careful to move the model with its parameters to the device we're using for calculations (a GPU if one is available, as on Google Colab).

In [ ]:
```
HIDDEN_DIMENSION = 4
model = FFNN(HIDDEN_DIMENSION).to(device)
```

We specify the criterion to be optimized as the mean square error loss function provided by PyTorch.

In [ ]:
```
criterion = nn.MSELoss(reduction='mean')
```

Evaluating data according to a model

To evaluate how well the model performs on some test data, we run the model forward on the x values and compute the loss relative to the y values. We define a function `eval_model` to carry out this calculation.

In [ ]:
```
def eval_model(model, criterion, data):
    """Applies the `model` to the x values in the `data` and returns the
       loss relative to the y values in the `data` along with the predicted
       y values."""
    model.eval()                          # turn on evaluation mode
    with torch.no_grad():                 # turn off propagating gradients
        X, Y = data                       # extract x and y values
        X = X.view(-1, 1).to(device)      # convert x and y to column vector
        Y = Y.view(-1, 1).to(device)      # ...and move them to the device
        predictions = model(X)            # calculate the predicted y values
        loss = criterion(predictions, Y)  # see how far off they are
    return loss.item(), predictions
```

All that remains is training the model. We'll use one of PyTorch's built in optimizers, the `Adam` optimizer. We set a few parameters for the training process: the learning rate, the number of "epochs" (passes through the training data) to perform, and the number of examples to train on at a time (the "batch size").

In [ ]:
```
## Parameters of the training regimen
LEARNING_RATE = 0.003
```

```
NUMBER_EPOCHS = 25
BATCH_SIZE = 20

## Choices for optimizers:

# Stochastic Gradient Descent (SGD) optimizer
# optim = torch.optim.SGD(model.parameters(), lr=learning_rate)

# The Adam optimizer, as described in the paper:
# Kingma and Ba. 2014. Adam: A Method for Stochastic Optimization.
# [https://arxiv.org/abs/1412.6980]
optim = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

Training the parameters of a model

Finally, we get to the function to train the parameters of the model so as to best fit the predictions to the actual values. We've provided the code, except for a few lines that you'll need to provide (marked `#TODO`), making use of some of the tools defined above. Those lines, which form the heart of the computation, calculate "forwards" to get the output predictions for the inputs, calculate the loss for those predictions, and calculate "backwards" the gradients of the loss for each of the parameters of the model. This sets up the optimizer to take a step of updating the parameters, making use of the calculated gradients to determine the direction to step. The saved gradients can then be zeroed and the process repeated.

> Note: The code we're asking you to write is *tiny*. If you find yourself writing more than a short line of code per `#TODO`, you're missing something.

In [ ]:

```
def train_model(model, criterion, optimizer, data,
                n_epochs=NUMBER_EPOCHS, batch_size=BATCH_SIZE):
    """Optimizes the parameters of the `model` by minimizing the `criterion`
       on the training `data`, using the `optimizer` algorithm for updates."
    model.train()                    # Turn on training mode

    X, Y = data
    trainX_len = len(X)

    loss_per_epoch = math.inf
    with tqdm(range(n_epochs)) as pbar:
        for epoch in pbar:
            pbar.set_postfix(epoch=epoch+1, loss=loss_per_epoch)
            loss_per_epoch = 0.
            for batch_i in range(int(trainX_len/batch_size)):
                optimizer.zero_grad()     # new batch; zero the gradients of

                # Input tensors and their corresponding output values for th
                batch_X = (X[batch_i * batch_size
                             : (batch_i+1) * batch_size] # extract examples
                           .view(-1, 1)                  # reshape to columr
                           .to(device)                   # move to device
                          )
```

```
                batch_Y = (Y[batch_i * batch_size
                             : (batch_i+1) * batch_size]
                           .view(-1, 1)
                           .to(device)
                          )

                #TODO: Calculate predictions for the x values in this batch
                predictions = ...

                #TODO: Calculate the loss for the predictions
                loss = ...

                #TODO: Perform backpropagation to calculate gradients
                ...

                # Update all parameters
                optimizer.step()

                loss_per_epoch += loss.item()
```

Putting it all together

Let's try it out. We start by generating some training and test data. The training data will be 10,000 samples of a noisy sinusoid. The test data, 100 samples from the same sinusoid, will be noise-free, so we can see how close the predictions are to noise-free outputs.

In [ ]:
```
train_data = sample_input(sinusoid, 10000, 5., frequency=1.5, noise=0.05)
test_data = sample_input(sinusoid, 100, 5., frequency=1.5)

plot_sample(train_data)
```

We train the model.

In [ ]:
```
model.reset_state()
train_model(model, criterion, optim, train_data)
```

...and test the trained model by evaluating it on the the test data.

In [ ]:
```
loss, predictions = eval_model(model, criterion, test_data)
```

In [ ]:
```
grader.check("model_reduces_loss")
```

We can see how well the model works by plotting the test data (circles) along with the predicted values (crosses).

In [ ]:
```
def visualize_predictions(data, predictions):
    X, Y = data

    # Plot the actual output values
    plt.plot(X.tolist(), Y.tolist(), '.', label = 'Target Values')
```

```
    # Plot the predicted output values
    predictions = predictions.view(-1, 1)
    plt.plot(X.tolist(), predictions.tolist(), 'x', label = 'Predictions')

    plt.xlabel('Input')
    plt.ylabel('Output')
    plt.legend()
    # we cannot use plt.show() because otter-grader does not support it
```

In [ ]:
```
# Visualize the predictions
visualize_predictions(test_data, predictions)
```

Trying different models

Now that we have the infrastructure, try experimenting with different models. Here are a few things you might play with. (No need to try them all.) What happens if you change the hidden dimension, increasing it to 8 or decreasing it to 2? What happens if you drop the middle layer? What about no middle layer but a much higher hidden dimension size? Does running for more epochs improve performance? Does the SGD optimizer work better or worse than the Adam optimizer?

Perform any experimentation in cells below this point, so you don't modify the cells above that are being unit tested.

Question: What conclusions have you drawn from your experimentation?

Type your answer here, replacing this text.

Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?
- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of lab 1-5 {-}

To double-check your work, the cell below will rerun all of the autograder tests.

In [ ]:
```
grader.check_all()
```