Entrée [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import random
import copy
from torch.autograd import Variable as V
```

Data generation

Let's create our tasks for sine, square, and triangle waves and potential noise in training sets.

Entrée [2]:

```
class Wave:
    def __init__(self, K, noise_percent):
        # K as in K-shot learning
        self.amp = np.random.uniform(0.1, 5.0)
        self.phase = np.random.uniform(0, np.pi)
        self.K = K
        self.noise_percent = noise_percent
        self.mini_train = None
        self.mini_test = None

    def mini_train_set(self):
        if self.mini_train is None:
            x = np.random.uniform(-5, 5, (self.K, 1))
            y = self.f(x) + np.random.normal(0, self.noise_percent*self.amp, (self.K
            self.mini_train=(x,y)
        return torch.Tensor(self.mini_train[0]), torch.Tensor(self.mini_train[1])

    def mini_test_set(self):
        self.mini_test = x = np.random.uniform(-5, 5, (self.K, 1))
        y = self.f(x)
        return torch.Tensor(x), torch.Tensor(y)

    def eval_set(self, size=50):
        x = np.linspace(-5, 5, size).reshape((size, 1))
        y = self.f(x)
        return torch.Tensor(x), torch.Tensor(y)

class SineWave(Wave):
    def __init__(self, K = 10, noise_percent = 0):
        super().__init__(K,noise_percent)

    def f(self, x):
        return self.amp * np.sin(x + self.phase)

class SquareWave(Wave):
    def __init__(self, K = 10, noise_percent = 0):
        super().__init__(K, noise_percent)

    def f(self, x):
        return self.amp * sp.square(x + self.phase)

class SawtoothWave(Wave):
    def __init__(self, K = 10, noise_percent = 0):
        super().__init__(K, noise_percent)

    def f(self, x):
        return self.amp * sp.sawtooth(x + self.phase)
```

Entrée [3]:

```
class DataGenerator:
    def __init__(self, function, size=50000, K = 10, noise_percent=0):
        self.size = size
        self.K = K
        self.function = function
        self.noise_percent = noise_percent
        self.tasks = None

    def generate_set(self):
        config = {"sine" : SineWave, "square" : SquareWave, "sawtooth" : SawtoothWav
        self.tasks = tasks = [config[self.function](self.K, self.noise_percent) for
        return tasks

    def shuffled_set(self):
        if self.tasks is None:
            self.generate_set()
        return random.sample(self.tasks, len(self.tasks))
```

Model creation

Entrée [5]:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(1, 40)
        self.hidden2 = nn.Linear(40, 40)
        self.out = nn.Linear(40, 1)

    def set_attr(self, name, param):
        #BIGGEST HACK EVER..
        if name == "hidden1.weight":
            self.hidden1.weight = param
        if name == "hidden1.bias":
            self.hidden1.bias = param
        if name == "hidden2.weight":
            self.hidden2.weight = param
        if name == "hidden2.bias":
            self.hidden2.bias = param
        if name == "out.weight":
            self.out.weight = param
        if name == "out.bias":
            self.out.bias = param

    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        return self.out(x)
```

Reptile Meta-Learning Algorithm

Entrée [6]:

```
class MetaLearner():
    def __init__(self, higher_order=False, lr_inner=0.01, lr_outer=0.001, sgd_steps_
        self.lr_inner = lr_inner
        self.lr_outer = lr_outer
        self.sgd_steps_inner = sgd_steps_inner
        self.higher_order = higher_order

    def inner_train(self, model, task, optimizer):
        optimizer.zero_grad()
        x, y = task.mini_train_set()
        predicted = model(x)
        loss = F.mse_loss(predicted, y)
        loss.backward(create_graph=keep_graph, retain_graph=keep_graph)
        optimizer.step()

    def init_grad(self, model):
        for param in model.parameters():
            param.grad = torch.zeros_like(param)

class Reptile(MetaLearner):
    def __init__(self, lr_inner=0.01, lr_outer=0.001, sgd_steps_inner=10):
        super().__init__(False, lr_inner, lr_outer, sgd_steps_inner)

    def compute_store_gradients(self, target, current):
        current_weights = dict(current.named_parameters())
        target_weights = dict(target.named_parameters())
        gradients = {name: (current_weights[name].data - target_weights[name].data)

        for name in current_weights:
            current_weights[name].grad.data = gradients[name]

    def train(self, model, train_data):
        optimizer = torch.optim.Adam(model.parameters(), lr=self.lr_outer)
        self.init_grad(model)

        for i, task in enumerate(train_data.shuffled_set()):
            optimizer.zero_grad()

            inner_model = copy.deepcopy(model)
            inner_optim = torch.optim.SGD(inner_model.parameters(), lr=self.lr_inner

            for _ in range(self.sgd_steps_inner):
                self.inner_train(inner_model, task, inner_optim)

            self.compute_store_gradients(inner_model, model)
            optimizer.step()

            if i % 5000 == 0:
                print("iteration:", i)
```

Entrée [116]:

```
reptile_model = Net()
reptile_learning_alg = Reptile()
train_data = DataGenerator("sine")
reptile_learning_alg.train(reptile_model, train_data)
```

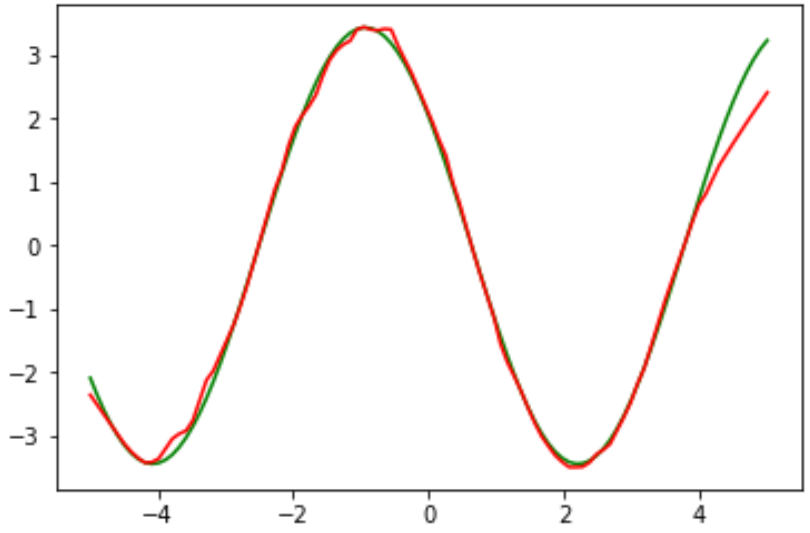
```
iteration: 0
iteration: 5000
iteration: 10000
iteration: 15000
iteration: 20000
iteration: 25000
iteration: 30000
iteration: 35000
iteration: 40000
iteration: 45000
```

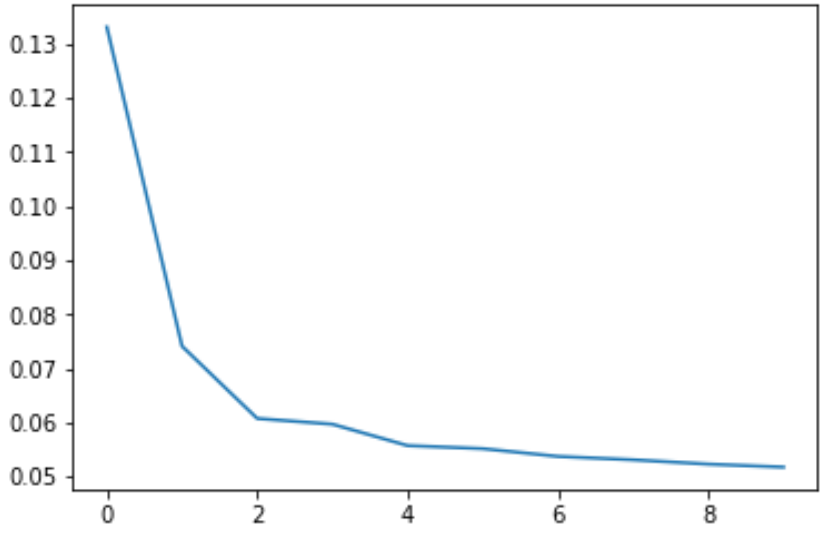
Entrée [7]:

```
def inner_train(model, task, optimizer):
    optimizer.zero_grad()
    x, y = task.mini_train_set()
    predicted = model(x)
    loss = F.mse_loss(predicted, y)
    loss.backward()
    optimizer.step()

def eval(model):
    test_task = SineWave()
#     xtest, ytest = test.mini_test_set()
    xeval, yeval = test_task.eval_set(size=100)

    model = copy.deepcopy(model)
    optim = torch.optim.SGD(model.parameters(), lr=0.01)
    losses = []

    for i in range(10):
        inner_train(model, test_task, optim)
        predicted = model(xeval)
        losses.append(F.mse_loss(predicted, yeval).item())

    xplot = xeval.numpy()
    yplot = yeval.numpy()
    predicted_plot = predicted.detach().numpy()

    plt.plot(xplot, yplot, 'g', label="ground truth")
    plt.plot(xplot, predicted_plot, 'r', label="predicted")
    plt.show()

    plt.plot(losses)
    plt.show()
```

Entrée [118]:

```
eval(reptile_model)
```





MAML

Entrée [39]:

```
class MAML(MetaLearner):
    def __init__(self, lr_inner=0.01, lr_outer=0.001, sgd_steps_inner=3):
        super().__init__(True, lr_inner, lr_outer, sgd_steps_inner)

    def inner_train(self, model, task):
        x, y = task.mini_train_set()
        predicted = model(x)
        loss = F.mse_loss(predicted, y)
        loss.backward(create_graph=True, retain_graph=True)
# #         grads = torch.autograd.grad(loss, model.parameters(), create_graph=True,
#         for (name, param), grad in zip(model.named_parameters(), grads):
#             model.set_attr(name, torch.nn.Parameter(param - self.lr_inner * grad))

    def train(self, model, train_data):
        optimizer = torch.optim.Adam(model.parameters(), lr=self.lr_outer)
        self.init_grad(model)

        for i, task in enumerate(train_data.shuffled_set()):
            inner_model = Net()

            for name, param in model.named_parameters():
                inner_model.set_attr(name, param)

            for _ in range(self.sgd_steps_inner):
                self.inner_train(inner_model, task)
                for name, param in inner_model.named_parameters():
#                     if i % 5000 == 0:
#                         print(param.grad)
                    inner_model.set_attr(name, torch.nn.Parameter(param - self.lr_in

            x, y = task.mini_test_set()
            predicted = inner_model(x)
            loss = F.mse_loss(predicted, y)
            loss.backward(retain_graph=True)
            if i % 5000 == 0:
                print("iteration:", i)
                for name, param in model.named_parameters():
                    print(param.grad.data)
                eval(model)
            optimizer.step()
            optimizer.zero_grad()
```

Entrée [40]:

```
maml_model = Net()
maml_learning_alg = MAML()
train_data = DataGenerator("sine")
maml_learning_alg.train(maml_model, train_data)
```

Entrée [42]:

```
eval(maml_model)
```