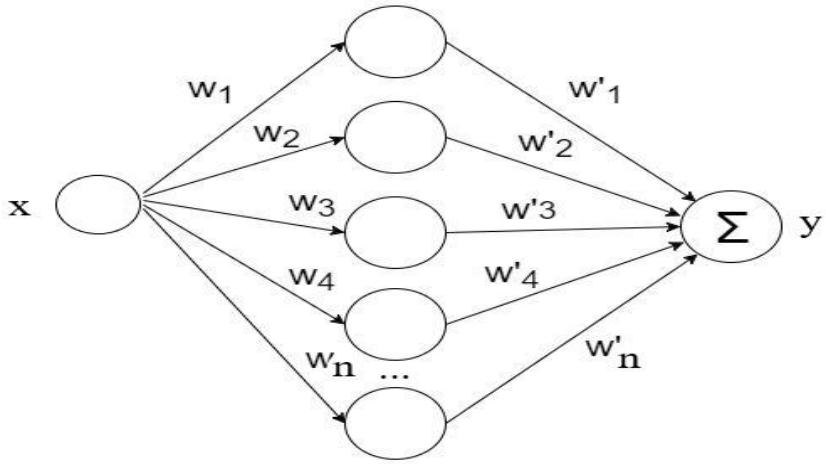
Лабораторная работа №1
по дисциплине «Системы поддержки принятия решений»
«Решение задачи регрессии с помощью нейронной сети»

Цель работы – исследование принципов разработки нейронной сети на примере задачи регрессии.

Регрессия – это односторонняя стохастическая зависимость, устанавливающая соответствие между случайными переменными, то есть математическое выражение, отражающее связь между зависимой переменной *y* и независимыми переменными *x* при условии, что это выражение будет иметь статистическую значимость.

Фактически, задача регрессии – *это предсказание некоторого вещественного числа*.

Для решения модельной задачи используем нейронную сеть с одним полносвязным скрытым слоем, представленную на рисунке.



Прежде чем рассматривать процесс обучения нейронной сети, мы рассмотрим *понятие размеченной обучающей выборки*.

Размеченная обучающая выборка состоит из какого-то количества объектов, для которых мы знаем две вещи: во-первых, это некоторые признаки x_1 ... x_N. Для каждого объекта мы знаем некоторый набор признаков. Кроме того, мы знаем некоторую метку объекта y_1 … y_N.

Мы можем на этом обучить некоторую нейронную сеть. Но, прежде чем обучать нейронную сеть, мы разделим эту выборку на три части*: тренировочный датасет, валидационный датасет и тестовый датасет*.

Тренировочный датасет – это то, что мы непосредственно используем для обучения нашей модели. Валидационный датасет нужен для того, чтобы подстраивать параметры обучения нашей модели (*гиперпараметры*). На самом деле его никогда не используют в процессе обучения, но мы подгоняем некоторые параметры, чтобы на этом датасете результаты были лучше. Тестовый датасет – это датасет, на котором мы будем проверять окончательный результат. Если у нас получится хороший результат на тестовом датасете, это означает, что наша модель обобщила информацию, которая ей была предоставлена.

Выполнение лабораторной работы можно производить в среде Google Colaboratory: https://colab.research.google.com/notebooks/welcome.ipynb.

Если вы хотите установить PyTorch на локальном компьютере, то можно установить дистрибутив Anaconda: https://www.anaconda.com/distribution/ с Python последней версии.

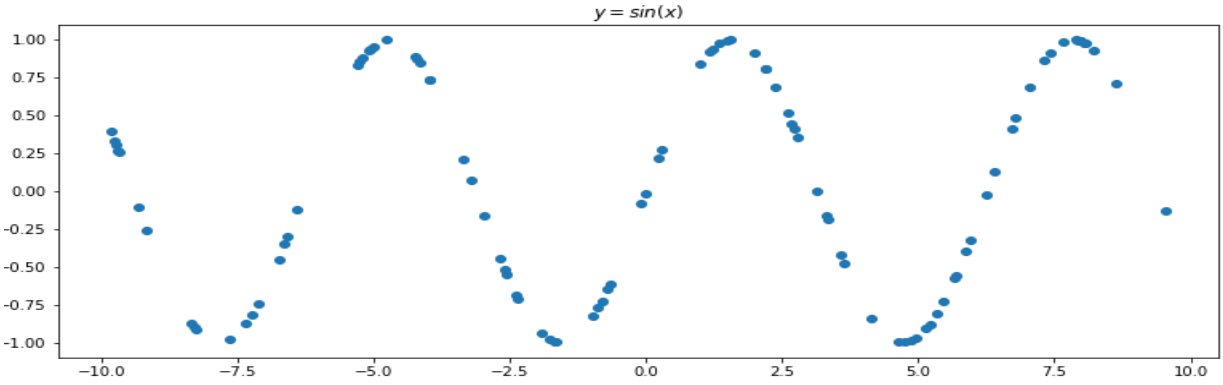
Общие положения

Рассмотрим следующую учебную задачу: предсказать функцию sin (*x*).

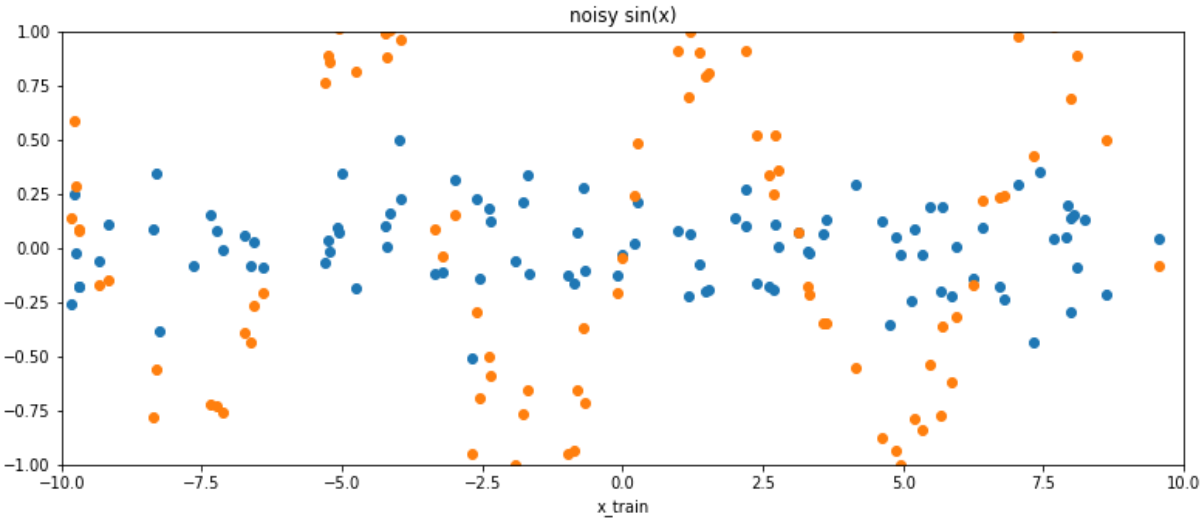
Текст программы представлен ниже.

1. Импортируем **torch** и **matplotlib**, чтобы рисовать графики.

2. Нужно составить "train dataset". Возьмём точки из равномерного распределения от нуля до единицы, 100 штук, каждую точку домножим на 20, отнимем от неё 10, чтобы график примерно был по центру – это будут наши значения "*x*". А "*y*" – это будут синусы от данных точек.



3. Добавим в обучающую выборку немного шума. Шум будет из нормального распределения. Этот шум прибавим к каждой точке предыдущего. Получится обучающая выборка.



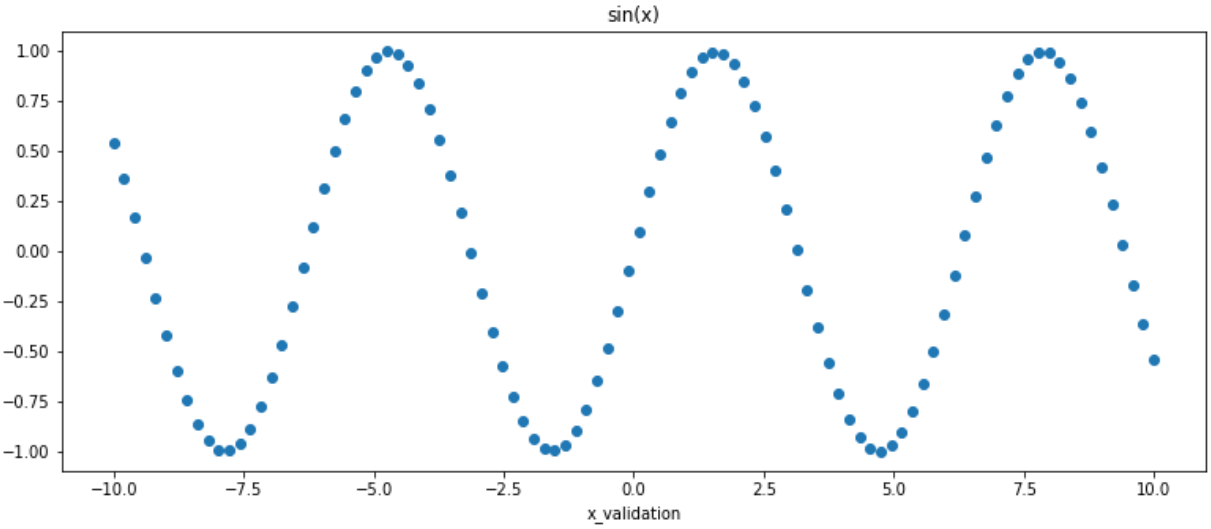
4. Когда мы будем передавать в нейросеть данные, нам хочется, чтобы они были правильной размерности. Ведь, по сути, признаком объекта может быть не одно число, как здесь (координата "*x*", по которой мы хотим предсказать координату "*y*"), а может быть сразу несколько чисел. Соответственно, для общности, нам нужно наш вектор *x* (который сейчас строчка), превратить в столбец, у которого в каждой строчке будет одно число *x*. Это делает метод unsqueeze(). Если вы в **PyTorch** видите нижнее подчёркивание в названии

метода, это значит, что этот метод трансформирует тот объект, к которому он применяется, то есть после выполнения этой ячейки у нас *x_train* и *y_train* изменились, и теперь это столбцы.

5. Кроме train dataset нам нужен будет отдельный validation dataset, мы делим наш dataset на тренировочные данные, и те, на которых мы будем тестировать или валидироваться. Сеть обучается на тренировочных данных и, соответственно, валидируется на тех данных которые она не видела.

Мы знаем, что тот закон природы, который сгенерировал наши данные – это функция синуса. Поэтому мы возьмём в валидационный датасет просто функцию синуса, не будем добавлять к ней никакого шума. Конечно это не очень жизненно, потому что у вас никогда такого не будет, что ваши данные будут не зашумлены, там всегда будет некоторый шум. Но в данном примере мы возьмём как валидацию обычный синус.

Создадим две переменные "*x_validation*" и "*y_validation*". Посмотрим, как выглядит график этой функции. Это обычный синус на точках, которые распределены равномерно от минус десяти до десяти. Мы будем их передавать в нейронную сеть, соответственно, они должны быть правильной размерности – это должен быть двумерный тензор, где каждая строчка соответствует одному элементу, одной точке. Делаем x_validation unsqueeze(1) и y_validation unsqueeze(1).



6. Теперь можно создать нейронную сеть. Чтобы создать нейронную сеть, нужно создать класс, назовём его **SineNet**, предполагая, что это будет нейросеть, которая решает задачу восстановления синуса. Её мы должны отнаследовать от класса **torch.nn.Module**. Такое наследование внесет в наш объект дополнительные функции, которые мы будем использовать. Кроме того, нужно проинициализировать те слои, которые будут использоваться в сети, с помощью функции "**__init__**", она на вход может принимать что угодно, любые параметры, которые нам будет интересно передать в эту сеть в момент конструирования. Например, интересно передать ***количество скрытых нейронов***, которые будут храниться в каждом слое, то есть мы предполагаем, что все слои будут одинакового размера. Там будет *n_hidden neurons*.

Теперь инициируем родительский объект. Создадим слои: первый слой, который будет называться **fc1**, это "fully connected" слой, полносвязный слой. В PyTorch "fully

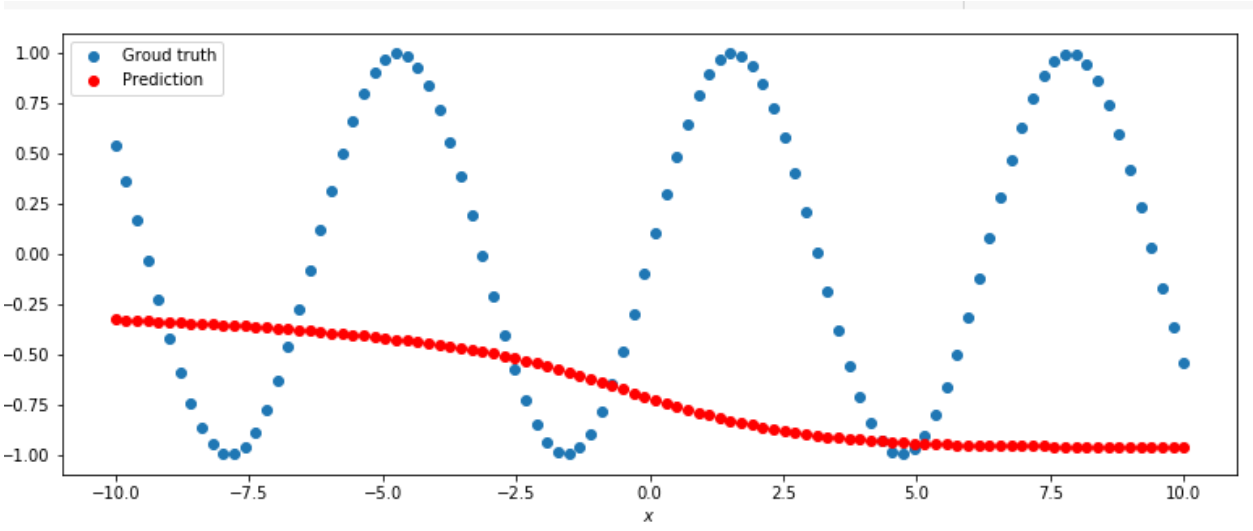
connected" слой называется "**linear**". Мы передаём количество входных нейронов и количество выходных нейронов. Входных нейронов будет ровно один. То есть это просто вход в нейрон. Это одно число "*x*", координата нашей точки, по которой мы будем что-то предсказывать. Выходных нейронов будет ровно "***n_hidden_neurons***".

7. После этого нужна функция активации. Кстати, попробуйте убрать функцию активации, узнайте, что получится. В качестве функции активации берем **сигмоиду**. В принципе, нам бы подошла любая функция активации. Но сигмоида – самая простая. Кроме того, мы добавим ещё один полносвязный слой, но у него будет всего один нейрон. Этот нейрон будет выходным - нашим ответом на вопрос о регрессии. Так как у нас задача регрессии, нас интересует ответ, который является одним числом, следовательно, на выходе нашей сети должен быть один нейрон. В итоге, наша нейросеть будет выглядеть, как два слоя, в одном из них будет несколько нейронов, а во втором будет один.

Теперь нам нужно написать функцию **forward**, описывающую как наши слои последовательно применяются. Сначала мы применяем слой "**fc1**" на "*x*". То, что получилось мы передаём в функцию активации, то что вышло из функций активации мы передаём в "**fc2**". В принципе, функция forward повторяет инициализацию.

Создадим такую сеть. Количество скрытых нейронов – 50, чтобы точно хватило. У нас теперь есть "SineNet" – объект, который можно обучать предсказанию.

8. Давайте не будем ничего обучать, а сразу предскажем. А вдруг уже сразу заработает? Напишем некоторую функцию "predict", внутри она будет очень простая – там будет вызов метода "forward". И если вы передадите туда некоторую переменную *x*, на выходе у вас будет некоторый prediction (одно искомое число). Далее есть некоторый код, который рисует этот prediction. Давайте посмотрим, что происходит. Представлено два графика, синим обозначен groud truth, (то, что мы бы хотели на валидации увидеть), *x* – это то, что мы передаём в сеть, а *y* – это то что мы бы хотели, чтобы сеть вернула, а красными точками обозначено то, что сеть нам предсказала.



Нетрудно догадаться, что, так как у нас сеть была инициирована случайными числами (*когда вы задаёте слои, они инициализируются некоторыми случайными числами*), то на выходе у нас получилась некоторая случайная кривая (она может быть разная в зависимости от запуска).

9. Давайте обучим эту нейросеть. Чтобы обучить нейросеть, нам нужно несколько вещей дополнительно. Во-первых, нам нужен некоторый оптимизатор – некоторый объект, который будет совершать для нас шаги градиентного спуска. Используем **torch.optim.adam**. Вы можете попробовать что-то другое. Метод ADAM – adaptive moment estimation – оптимизационный алгоритм. Он сочетает в себе идею накопления движения и идею более слабого обновления весов для типичных признаков (наиболее известен SGD – обычный градиентный спуск, но в данной задаче он работает не очень хорошо, собственно, поэтому используем другие градиентные спуски).

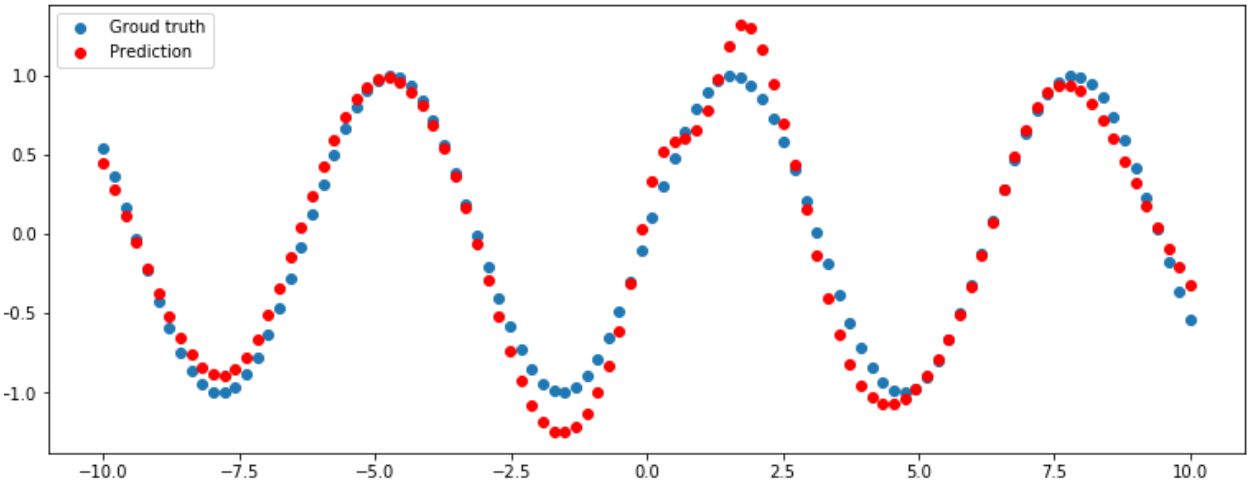
Очень важно, что на вход Adam передаются те параметры, которые мы хотим модифицировать – параметры, которые мы хотим обучать в нейронной сети. Можно было подумать, что это "*x*", но это не так, потому что "*x*" – это наши точки, мы не можем на них никак повлиять. Зато мы можем *повлиять на веса нейронной сети – те веса, которые хранятся в нейронах* (они находятся в **sine_net.parameters**). Это одна из тех причин, почему нейросеть мы наследовали, а не создавали как класс с нуля. Соответственно, если мы передадим в Adam такой объект, то ADAM поймёт, что здесь лежат все те переменные, которые он может модифицировать вследствие градиентного спуска. Также необходимо передать "**learning rate**" – шаг градиентного спуска. Здесь взято значение **0.01**, можно также попробовать другие значения.

10. Создадим такой оптимизатор. Также нам нужна функция потерь – та функция, которая говорит, насколько неправильно мы предсказали, насколько мы ошиблись. Это та функция, по которой будет происходить *вычисление градиента,* и которая будет участвовать в градиентном спуске – функция **loss function**. В данной работе используется **MSE (mean squared error**).

11. Начнем тренировку. Если мы возьмём весь наш датасет, прогоним его через нейросеть, получим некоторые предсказания. После этого, на этих предсказаниях посчитаем функцию потерь, которую мы задали. После этого у функции потерь посчитаем производную и сделаем градиентный шаг. Это будет называться **эпохой**. Таким образом, мы посмотрели на все наши данные и сделали градиентный шаг.

Возможно нам потребуется много эпох. Тут взято две тысячи, чтобы точно хватало. Что мы делаем внутри одной итерации, внутри одной эпохи? Сначала мы обнуляем градиенты. Можно было обнулять градиенты в конце, но, чтобы не забыть, лучше сразу делать это вначале.

Каждая эпоха начинается с того, что у оптимайзера обнуляются градиенты. После этого считаем forward, то есть мы берём наш весь X_train и передаём его в функцию forward, считаем prediction (считаем предсказания нашей нейросети), после этого считаем функцию потерь, получаем некоторое число – скаляр, по которому мы можем сделать backward. Делаем по этому скаляру backward, то есть это некоторый тензор, который зависит от параметров сети, то есть от весов нейросети, который обернут в оптимайзер, и, соответственно, когда мы делаем loss_val.backward, оптимайзер понимает, что там посчитались градиенты, и значит оптимайзер может сделать шаг. Это весь цикл обучения на одной эпохе. Давайте проведём обучение на 2000 эпохах и посмотрим, что получится.



Мы видим, что результат хороший. Мы обучались на зашумлённых данных, которые совсем не похожи на синусы. Получили некоторые точки, которые здесь отмечены красным. Оказывается, что у нас довольно хорошие результаты, мы обучались на зашумлённых данных, а получили действительно функцию, которая очень близка к синусу. Она не идеальный синус потому что данные, которые приходили, объективно были не очень похожи на идеальный синус.

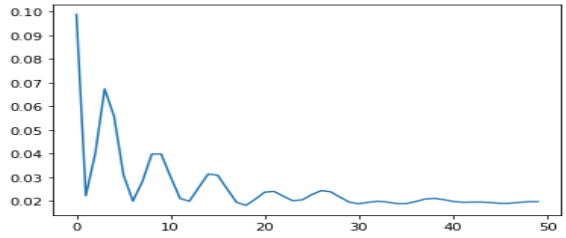
12. Посмотрим, как можно улучшить результаты. Во-первых, мы могли взять и поставить поменьше нейронов в нашем единственном скрытом слое. Поставим, например, один нейрон. Что будет в таком случае? Передаём параметры нейросети в оптимайзер. Можно ещё раз инициировать loss function, это ни на что не влияет. И обучимся заново. Можно увидеть, что если всего один нейрон, то у нас получается практически линейное предсказание. Наверное, одного нейрона недостаточно, наша нейросеть не очень сложная, она не может предсказать сложную функцию.

Давайте добавим ещё нейронов (например, 3), у нас получится уже один изгиб. Можно заметить интересную закономерность: чем больше нейронов мы используем в скрытом слое, тем больше изгибов у нашей итоговой функции получается. Если вы представите эту нейросеть, то это простая сумма сигмоид. В нашем скрытым слое есть N сигмоид.

Это частый случай, когда создана нейронная сеть более сложная, чем требуется, которая может аппроксимировать более сложные функции, чем та функция, которую мы имеем. Переусложнение нейронной сети – это не всегда плохо.

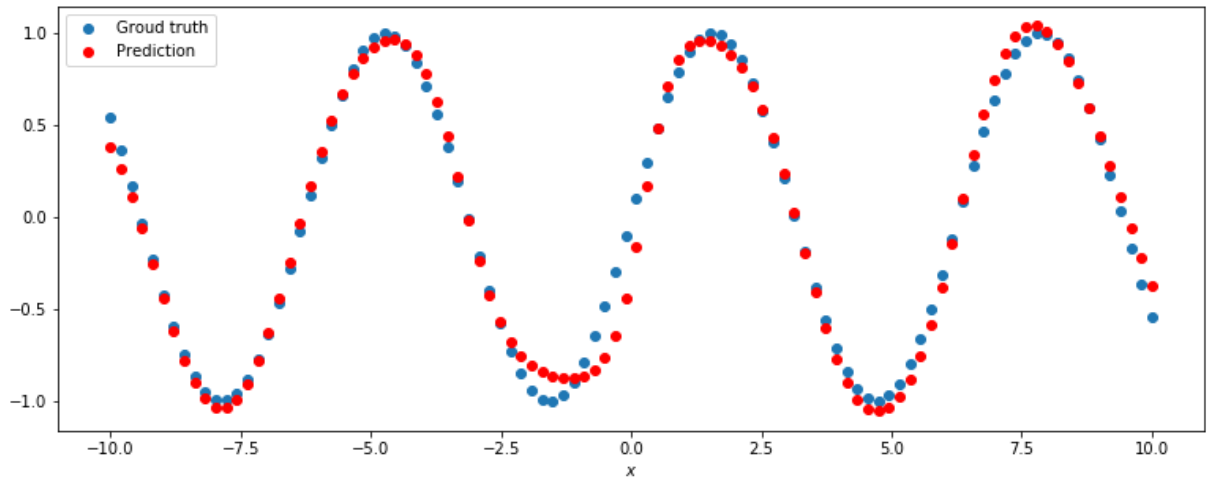
14. Для визуализации процесса обучения по эпохам можно добавить код, показанный красным жирным шрифтом. Вид функции ошибки по эпохам (для 50 эпох) показан на графике.

```
plt.plot(test_loss_history);
```



```
##
import torch
import matplotlib.pyplot as plt
##

##
import matplotlib
matplotlib.rcParams['figure.figsize'] = (13.0, 5.0)
##


##
x_train = torch.rand(100)
x_train = x_train * 20.0 - 10.0
y_train = torch.sin(x_train)
plt.plot(x_train.numpy(), y_train.numpy(), 'o')
plt.title('$y = sin(x)$');
##


##
noise = torch.randn(y_train.shape) / 5.
plt.plot(x_train.numpy(), noise.numpy(), 'o')
plt.axis([-10, 10, -1, 1])
plt.title('Gaussian noise');
y_train = y_train + noise
plt.plot(x_train.numpy(), y_train.numpy(), 'o')
plt.title('noisy sin(x)')
plt.xlabel('x_train')
plt.ylabel('y_train');
##


##
x_train.unsqueeze_(1)
y_train.unsqueeze_(1);
##


##
x_validation = torch.linspace(-10, 10, 100)
y_validation = torch.sin(x_validation.data)
plt.plot(x_validation.numpy(), y_validation.numpy(), 'o')
plt.title('sin(x)')
plt.xlabel('x_validation')
plt.ylabel('y_validation');

x_validation.unsqueeze_(1)
y_validation.unsqueeze_(1);
##


##
class SineNet(torch.nn.Module):
    def __init__(self, n_hidden_neurons):
        super(SineNet, self).__init__()
        self.fc1 = torch.nn.Linear(1, n_hidden_neurons)
        self.act1 = torch.nn.Sigmoid()
        self.fc2 = torch.nn.Linear(n_hidden_neurons, 1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        return x

sine_net = SineNet(50)
##
```

```
##
def predict(net, x, y):
    y_pred = net.forward(x)

    plt.plot(x.numpy(), y.numpy(), 'o', label='Groud truth')
    plt.plot(x.numpy(), y_pred.data.numpy(), 'o', c='r', label='Prediction');
    plt.legend(loc='upper left')
    plt.xlabel('$x$')
    plt.ylabel('$y$')

predict(sine_net, x_validation, y_validation)
##


##
optimizer = torch.optim.Adam(sine_net.parameters(), lr=0.01)
##

#test_accuracy_history = []
#test_loss_history = []

##
def loss(pred, target):
    squares = (pred - target) ** 2
    return squares.mean()
##


##
for epoch_index in range(2000):
    optimizer.zero_grad()

    y_pred = sine_net.forward(x_train)
    loss_val = loss(y_pred, y_train)

    loss_val.backward()

    optimizer.step()

#   test_preds = sine_net.forward(x_validation)
#   test_loss_history.append(loss(test_preds, y_validation))

predict(sine_net, x_validation, y_validation)
##


##
#plt.plot(test_loss_history);
##
```



Задание на лабораторную работу

1. Исследовать нейронную сеть при заданных начальных параметрах (см. таблицу). Найти минимальное значение ***n_hidden_neurons***, при котором сеть дает удовлетворительные результаты.
2. Найти наилучшее значение шага градиентного спуска ***lr*** в интервале $\pm$ 100% от номинального значения.
3. Изменить нейронную сеть для предсказания функции $y = 2^x * sin(2^{-x})$
4. Для этой задачи (п. 3) получите **метрику** MAE = $\frac{1}{l}\sum_i^l |y\_pred_{\ i} - y\_target_{\ i}|$ не хуже **0.03**, варьируя: архитектуру сети, loss-функцию, *lr* оптимизатора или количество эпох в обучении.
5. Метрика вычисляется с помощью выражения **(pred - target).abs().mean()** и выводится оператором

print(metric(sine_net.forward(x_validation), y_validation).item()).

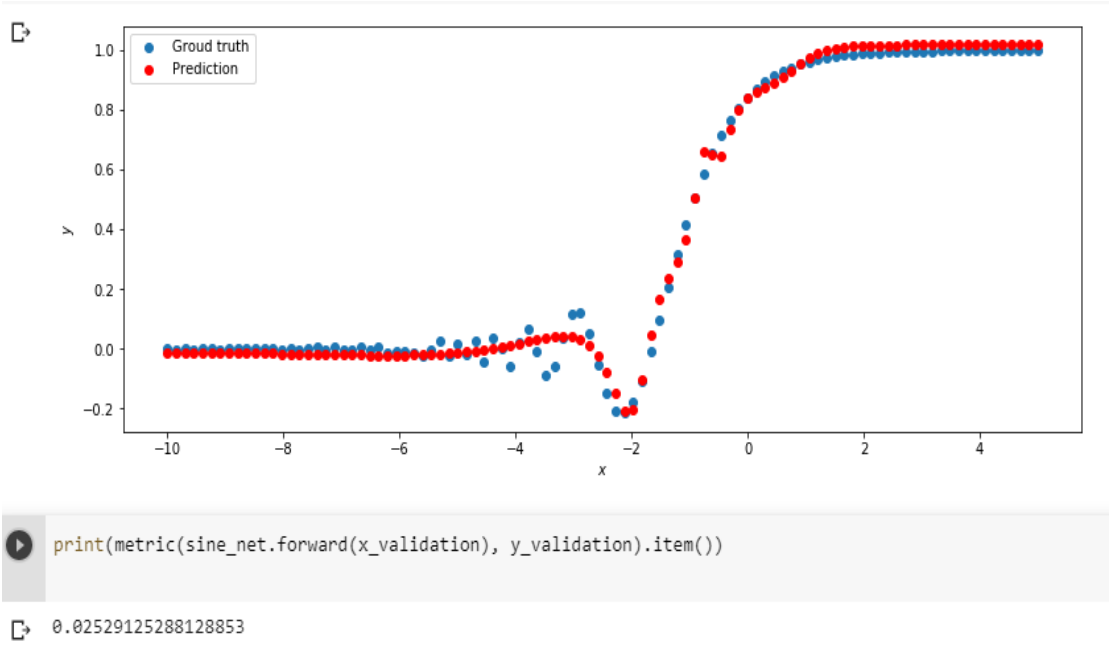


Таблица. Начальные значения гиперпараметров нейронной сети

Вариант	**Метод оптимизации**	**Число нейронов в скрытом слое *n_hidden_neurons***	**Шаг градиентного спуска *lr***
0	ADAM	10	0.01
1	ADAM	20	0.001
2	ADAM	30	0.01
3	ADAM	40	0.001
4	ADAM	50	0.01
5	SGD	10	0.001
6	SGD	20	0.01
7	SGD	30	0.001
8	SGD	40	0.01
9	SGD	50	0.001

Примечание

1. **Подготовка данных для функции $y = 2^x * sin(2^{-x})$**

```
def  target_function(x):
      return 2**x * torch.sin(2**-x)

# ------Dataset preparation start--------
x_train =  torch.linspace(-10, 5, 100)
y_train = target_function(x_train)
noise = torch.randn(y_train.shape) / 20.
y_train = y_train + noise
x_train.unsqueeze_(1)
y_train.unsqueeze_(1)

x_validation = torch.linspace(-10, 5, 100)
y_validation = target_function(x_validation)
x_validation.unsqueeze_(1)
y_validation.unsqueeze_(1)
# ------Dataset preparation end--------
```

2. **Описание функции metric**

```
def metric(pred, target):
    return (pred - target).abs().mean()
```

Содержание отчета

1. Титульный лист
2. Цель работы, постановка задачи исследования.
3. Описание методики исследования.
4. Результаты исследования в соответствии с заданием.
5. Выводы по работе.