

Лабораторная работа

Тема: нейронные сети

Цель работы: получение практических навыков программирования нейронных сетей на языке Python с использованием библиотеки PyTorch.

Задание: используя программу Jupiter Notebook, язык программирования Python, библиотеку PyTorch построить нейронную сеть по варианту и использовать для получения результата.

Работа заключается в:

- Загрузке / генерации данных для обучения НС;
- Построения НС;
- Обучения НС;
- Проверки НС на тестовых данных;
- Визуализация результата.

Отчёт по лабораторной работе должен содержать:

1. Фамилию и номер группы учащегося, задание, вариант
2. Схему НС (ее слоёв)
3. Описание входных данные
4. Описание алгоритма обучения с учетом варианта (функции потерь, оптимизатора и т.д.)
5. Графики динамики обучения НС.
6. Результат тестирования НС.
7. Код.

Методические указания по использованию библиотеки PyTorch для построения нейронных сетей

Пакет torch.nn

Пакет `torch.nn` используется для создания нейронных сетей. Он содержит контейнеры для НС, в котором определяются слои, функции потерь, активации, различные методы оптимизации для реализации обучения и т.д.

Пакет `nn` определяет набор модулей, которые примерно эквивалентны слоям нейронной сети. Модуль принимает входные Tensors и вычисляет выходные Tensors, но может также содержать внутреннее состояние, такое как Tensors, содержащее обучаемые параметры.

Алгоритм работы с НС

- 1 Подготовка данных для обучения /анализа (обучающая выборка), их преобразование.
- 2 Выбирается тип НС в зависимости от поставленной задачи (прямого распространения, рекуррентная, сверочная и т.д.), выбирается архитектура сети (количество слоёв, нейронов в слоях, типы слоёв, функции активации), строится модель.
- 3 Определение функции потерь.
- 4 Определение оптимизатора.
- 5 Цикл обучения НС (на примере сети прямого распространения)
 - ввод данных и вычисление результата (прямой проход)
 - вычисление потери (насколько далёк результат от правильности).
 - градиентный спуск и коррекция весов (обратный проход).
- 6 Запуск / тестирование НС (на тестовой выборке).

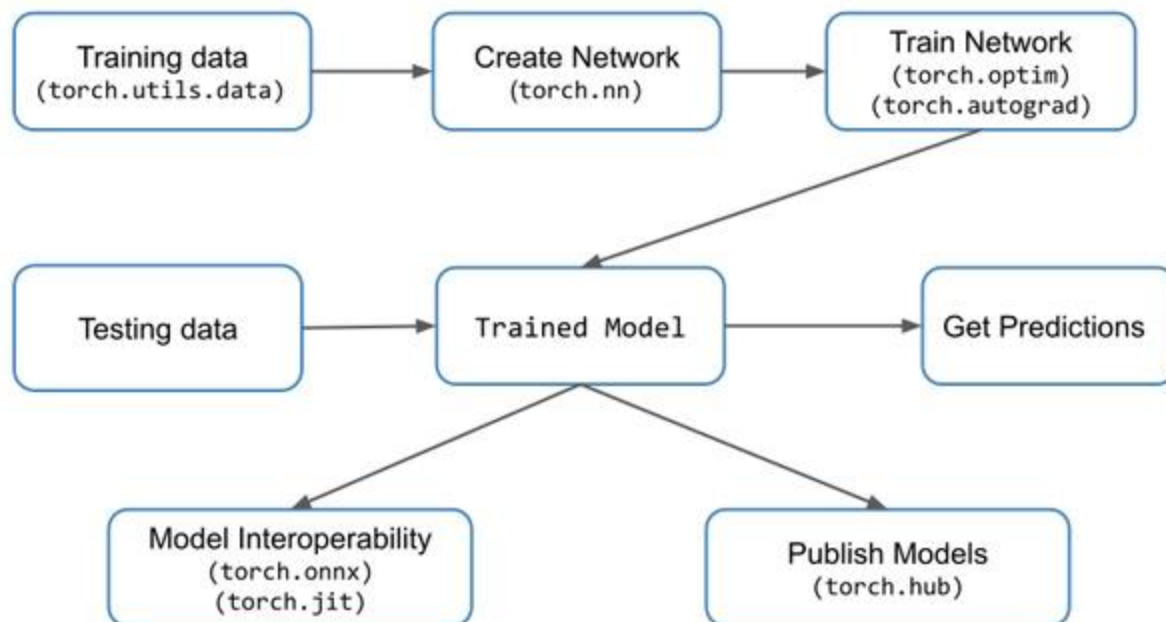


Рисунок 1 – Схема применения модулей PyTorch для обучения НС¹

¹ https://ai-news.ru/2019/07/pytorch_dlya_nachinaushih_osnovy.html

Загрузка подготовленного набора данных

PyTorch включает в себя пакет torchvision, который используется для загрузки и подготовки набора данных (<https://pytorch.org/docs/master/torchvision/index.html>). Он включает в себя две основные функции, а именно Dataset и DataLoader, которые помогают в преобразовании и загрузке набора данных.

Dataset построен поверх тензорного типа данных и используется в основном для пользовательских наборов данных. Набор данных используется для чтения и преобразования точки данных из данного набора данных.

Dataset - абстрактный класс, представляющий набор данных. Пользовательский набор данных должен наследовать Dataset и переопределять следующие методы:

- `__len__`, чтобы `len(dataset)` возвращал размер набора данных.
- `__getitem__` для поддержки индексации, так что `dataset[i]` может использоваться для получения i-го экземпляра

Основной синтаксис для реализации упомянут ниже:

```
trainset = torchvision.datasets.CIFAR10(root = './data', train = True,
    download = True, transform = transform)
```

Пример:

1) Загрузка набора данных MNIST

```
import torchvision
train_dataset = torchvision.datasets.MNIST(root='g:\\DataForNN2', train=True,
    transform=False, download=True)
```

3) Загрузка набора CIFAR10

```
trainset = torchvision.datasets.CIFAR10(root = DATA_PATH, train = True,
    download = True, transform = False)
```

4) Загрузка набора STL10

```
torchvision.datasets.STL10(DATA_PATH, split='train', folds=None,
    transform=None, target_transform=None, download=True)
```

Далее необходимо создать объекты `train_dataset` и `test_dataset`, которые будут последовательно проходить через загрузчик данных. Чтобы создать такие датасеты из данных MNIST, требуется задать несколько аргументов. Первый — путь до папки, где хранится файл с данными для тренировки и тестирования. Логический аргумент `train` показывает, какой файл из `train.pt` или `test.pt` стоит брать в качестве тренировочного сета. Следующий аргумент — `transform`, в котором мы указываем ранее созданный объект `trans`, который осуществляет преобразования. Наконец, аргумент загрузки просит функцию датасета MNIST загрузить при необходимости данные из онлайн источника.²

Таблица 1 – Примеры наборов данных для обучения (полный список см. <https://pytorch.org/docs/stable/torchvision/datasets.html>)

MNIST	Рукописные цифры 1–9. Подмножество набора данных NIST рукописных символов. Содержит обучающий набор из 60000 тестовых изображений и тестовый набор из 10000.
Fashion-	Набор данных для MNIST. Содержит изображения предметов моды;

² <https://neurohive.io/ru/tutorial/cnn-na-pytorch/>

MNIST	например, футболка, брюки, пуловер.
EMNIST	На основе рукописных символов NIST, включая буквы и цифры и разделение для задач классификации классов 47, 26 и 10.
COCO	Более 100 000 изображений, классифицированных в повседневные предметы; например, человек, рюкзак и велосипед. Каждое изображение может иметь более одного класса.
LSUN	Используется для крупномасштабной классификации сцен изображений; например, спальня, мост, церковь.
Imagenet-12	Крупномасштабный набор данных визуального распознавания, содержащий 1,2 миллиона изображений и 1000 категорий. Реализовано с классом ImageFolder, где каждый класс находится в папке.
CIFAR	60 000 цветных изображений с низким разрешением (32 32) в 10 взаимоисключающих классах; например, самолет, грузовик и автомобиль.
STL10	Аналогично CIFAR, но с более высоким разрешением и большим количеством немаркированных изображений.
SVHN	600 000 изображений номеров улиц, полученных из Google Street View. Используется для распознавания цифр в реальных условиях.
PhotoTour	Изучение локальных дескрипторов изображений. Состоит из полутоновых изображений, состоящих из 126 фрагментов, сопровождаемых текстовым файлом дескриптора. Используется для распознавания образов.

Преобразование данных

Функция `transform.Compose()` находится в пакете `torchvision` и позволяет выполнять трансформацию набора данных, причём трансформаций может быть несколько и они представляются списком.

Пример:

1) Загрузка MNIST с преобразованием

```
import torchvision
import torchvision.transforms as transforms
# путь куда грузим
DATA_PATH = 'g:\\DataForNN'
# выполняемое преобразование над набором данных
trans = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.1307,), (0.3081,))])
# грузим набор данных тренировочный
train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=True,
transform=trans, download=True)
# грузим набор данных тестовый
test_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=False,
transform=trans)
```

В примере устанавливается преобразование, которое конвертирует входной датасет в PyTorch тензор. PyTorch тензор — особый тип данных, используемый в библиотеке для всех различных операций с данными и весами внутри нейросети. Следующий аргумент в списке `Compose()` — нормализация. Нейронная сеть обучается лучше, когда входные данные нормализованы так, что их значения находятся в диапазоне от -1 до 1 или от 0 до 1. Чтобы это сделать с помощью нормализации PyTorch, необходимо указать среднее и стандартное отклонение MNIST датасета, которые в этом случае равны 0.1307 и 0.3081 соответственно. У MNIST есть только один канал, но уже для датасета CIFAR с 3

каналами (по одному на каждый цвет из RGB спектра) надо указывать среднее и стандартное отклонение для каждого.

Загрузка данных для тренировки нейронной сети

`DataLoader` используется, когда есть большой набор данных, и необходимо загрузить данные из `Dataset` в фоновом режиме, чтобы он был готов и ждал цикла обучения.

`DataLoader` используется для перемешивания и пакетной обработки данных. Он может использоваться для загрузки данных параллельно с многопроцессорными рабочими. Объект загрузчик данных в `PyTorch` обеспечивает несколько полезных функций при использовании тренировочных данных:

- Возможность легко перемешивать данные.
- Возможность группировать данные в партии.
- Более эффективное использование данных с помощью параллельной загрузки, используя многопроцессорную обработку.

Синтаксис:

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size = 4,
                                           shuffle = True)
```

Первый — данные, которые вы хотите загрузить; второй — желаемый размер партии; третий — перемешивать ли случайным образом датасет.

Построение нейронной сети

На базе `nn.Module` . Необходимо наследовать класс `nn.Module` и реализовать методы инициализации `init` и прямого прохода/вычисления `forward` . Синтаксис следующий:

```
# подключаем модуль torch.nn
import torch.nn as nn
# импортируем функции активации
import torch.nn.functional as F
# Model это имя
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        ...
    def forward(self, x):
        ...
```

Внутри функции `__init__` объявляют слои будущей нейронной сети, они могут быть разными по типу, в зависимости от того, какую нейронную сеть строим, количество и размерность слоёв определяется здесь же. Размерность следующих друг за другом слоев должна быть согласована, сколько выходов в предшествующем, столько входов в последующем.

Пример:

1) Два линейных слоя (нейронная сеть прямого распространения, в которой слои полносвязные), которые имеют вид $\mathbf{W}'\mathbf{x}+\mathbf{b}$, где \mathbf{W} — матрица весов размером $(input, output)$ и \mathbf{b} — вектор смещения размером $output$. Первый слой

размерностью (784, 100), второй (100, 10), т.е. выходной вектор НС размерностью 10:

```
class Model(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 100)
        self.fc2 = nn.Linear(100, 10)
```

2) Два слоя двумерной свёртки, первый слой имеет 1 входной канал, 20 выходных и размер ядра 5, второй, 20 входных :

```
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)
```

3) Свёрточная сеть с 2 сверточными слоями и 3 линейными (полносвязными):

```
class Model(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.fc1 = nn.Linear(16 * 6 * 6, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

4) Свёрточная сеть Conv2d -> MaxPool2d -> Conv2d -> MaxPool2d -> Linear -> Linear.

```
class MNISTConvNet(nn.Module):
    def __init__(self):
        super(MNISTConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(10, 20, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
```

5) Рекуррентная нейронная сеть.

```
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.layer_dim = layer_dim
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True,
                           nonlinearity='relu')
        self.fc = nn.Linear(hidden_dim, output_dim)
```

Метод *forward* используется непосредственно для преобразования входных данных с помощью заданной нейронной сети в ее выходы.

Вычисляемая функция может быть любой сложности, но должна учитывать заданные слои в функции *init*.

Пример:

1) Определение для линейных слоёв из примеров выше. Функция *view()* переиндексирует тензор с данными заданным образом, "-1" в качестве первого аргумента функции означает, что количество элементов в первой размерности будет вычислено автоматически. Если исходный тензор *x* имеет размерность (N, 28, 28), то после *x = x.view(-1, 28*28)* его размерность станет равна (N, 784).

```
def forward(self, x):
    x = x.view(-1, 28*28)
```

```

x = F.relu(self.fc1(x))
x = self.fc2(x)
x = F.softmax(x, dim=1)
return x

```

2) Для 2 примеры выше, обращаемся по определённым ранее именам слоёв, используется функция relu:

```

def forward(self, x):
    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))

```

3) Пример для сверточной сети

```

def forward(self, x):
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    x = x.view(-1, self.num_flat_features(x))
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x

```

4) Пример для свёрточной сети Conv2d -> MaxPool2d -> Conv2d -> MaxPool2d -> Linear -> Linear

```

def forward(self, input):
    x = self.pool1(F.relu(self.conv1(input)))
    x = self.pool2(F.relu(self.conv2(x)))
    x = x.view(x.size(0), -1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    return x

```

4) Пример для рекуррентной сети.

```

def forward(self, x):
    h0 = Variable(torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim))
    out, hn = self.rnn(x, h0)
    out = self.fc(out[:, -1, :])
    return out

```

Для построения модели надо создать объект описанного класса:

```
Net=Model()
```

На базе *nn.Sequential*. Контейнер для линейных / последовательных слоёв *Linear*, которые имеют вид $\mathbf{W}'\mathbf{x}+\mathbf{b}$, где \mathbf{W} — матрица весов размером (*input*, *output*) и \mathbf{b} — вектор смещения размером *output*.

Используя этот контейнер можно в одном операторе определить и вид НС и как будут вычисляться выходные значения.

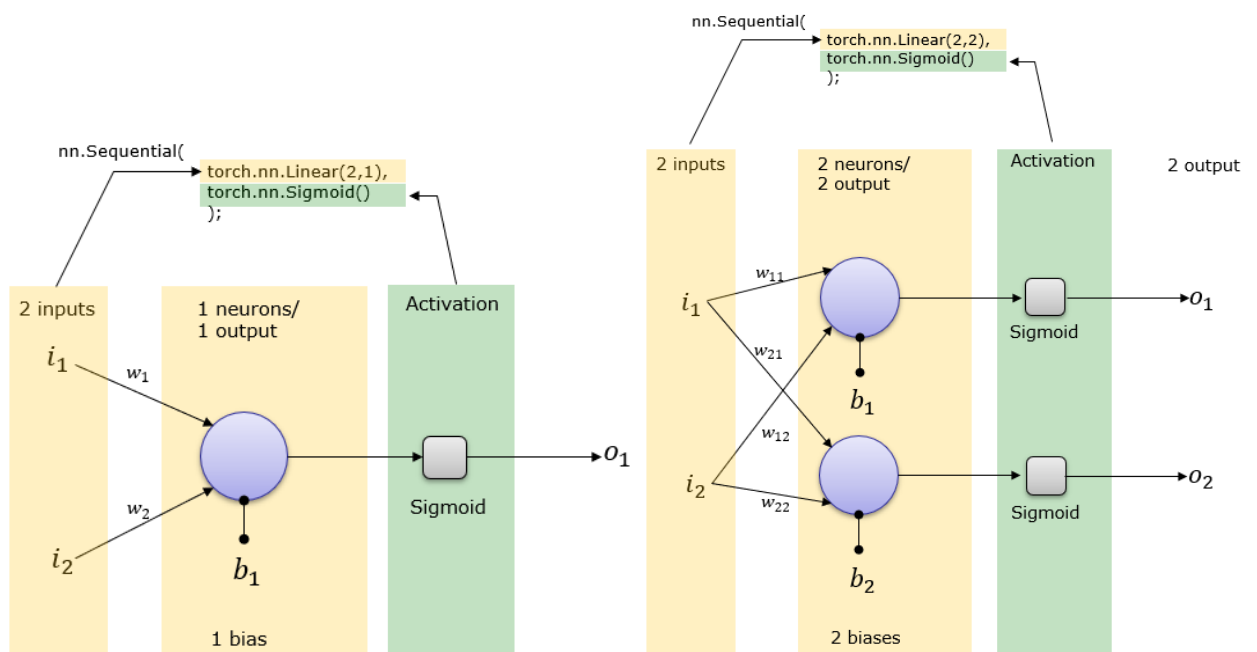
Пример:

НС с 10 входами, с функцией ReLU(), 5 нейронов в скрытом слое, выходной нейрон 1 с функцией сигмоида.

```

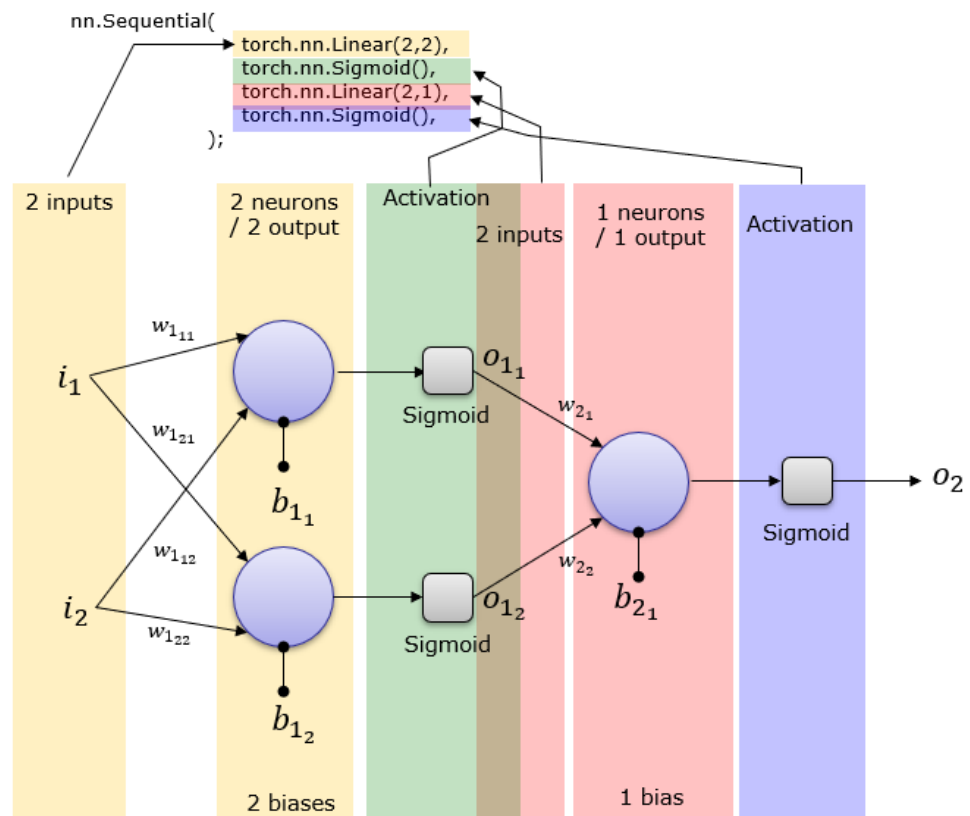
model = nn.Sequential(nn.Linear(10, 5),
    nn.ReLU(),
    nn.Linear(5, 1),
    nn.Sigmoid())

```



а) из 2 входов и 1 выхода

б) 2 входа 2 выхода



в) 2 слоя в первом 2 нейрона во втором 1, функция активации - сигмоида

Рисунок 2 - Примеры простейшего линейного слоя³

Существуют ещё другие возможности по построению НС. В зависимости от вида НС слои (Таблица 2) и функции активации (

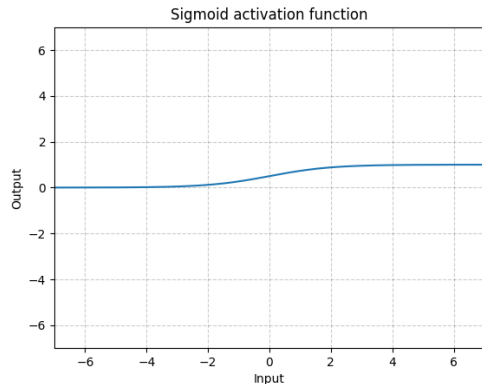
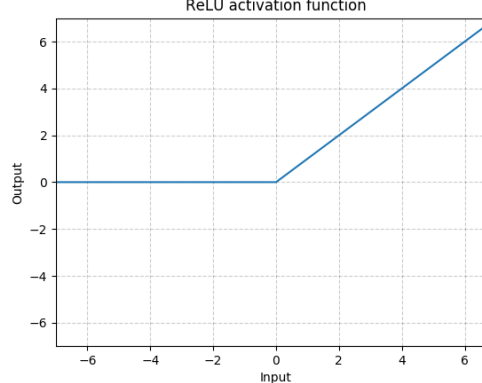
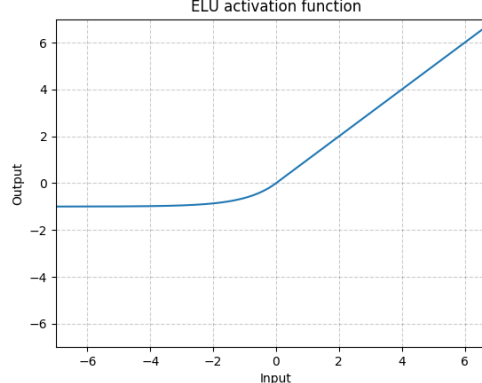
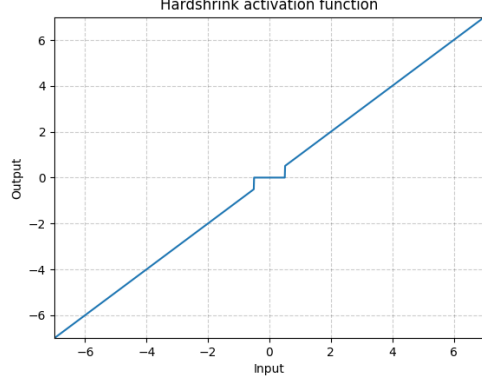
³ http://www.sharetechnote.com/html/Python_PyTorch_nn_Sequential_01.html

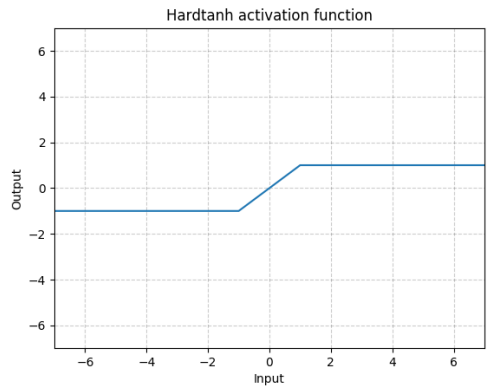
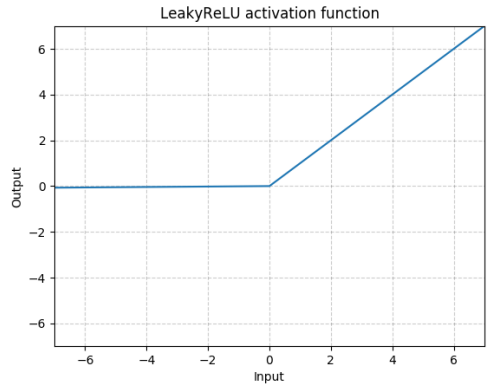
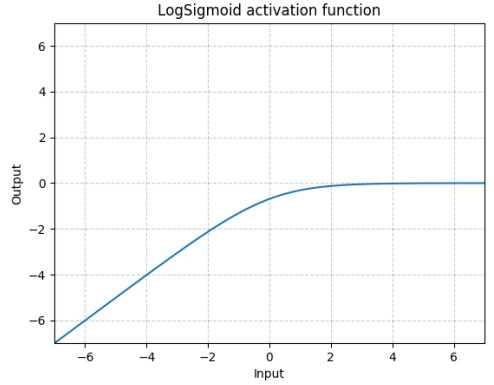
Таблица 3) могут быть разными., см. подробнее <https://pytorch.org/docs/stable/nn.html>.

Таблица 2 – Примеры типов слоёв НС

Типы слоёв	Назначение	Описание
Активация (activation)	Общие для НС	Содержит функцию активации, которую применяют ко входам слоя.
Нормализация (Normalization)		Слой обеспечивает применение градиентного спуска не к одной точке выборки, а к небольшой коллекции данных.
Прореживание (dropout), регуляризация		Слой обеспечивает добавление информации к условию с целью предотвращения переобучения.
Рекуррентные (Recurrent)	Для рекуррентных НС	Основной блок рекуррентных НС
Свёртка (Convolution)	Для свёрточных НС	основной блок свёрточной нейронной сети, включает в себя для каждого канала свой фильтр, ядро свёртки которого обрабатывает предыдущий слой по фрагментам (суммируя результаты поэлементного произведения для каждого фрагмента).
Пулинг / группировка / субдискретизация (Pooling)		слои пулинга, как правило, чередуются со слоями свёртки и обобщает (упрощает) информацию, полученную от слоя свёртки, пулинг «сжимает» карты признаков, полученные на предыдущем сверточном слое.
Дополнение отступа (Padding)		пиксели, которые находятся на границе изображения участвуют в меньшем количестве сверток, чем внутренние. В связи с этим в сверточных слоях используется дополнение изображения (англ. padding). Выходы с предыдущего слоя дополняются пикселями так, чтобы после свертки сохранился размер изображения. Такие свертки называют одинаковыми (same convolution), а свертки без дополнения изображения называются правильными (valid convolution).
Линейный / Соединение «все-со-всеми» / полносвязный (Linear)	Для НС с прямым распространением и свёрточных	Все входы связаны со всеми нейронами слоя, используются в НС прямого распространения, как последний слой свёрточной сети.

Таблица 3 – Пример функций активации нейронов (подробнее см. <https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity>)

Функция активации	Метод функции / формула	График функции
Sigmoid	<code>torch.nn.Sigmoid()</code> $\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)}$	 <p>Sigmoid activation function</p>
ReLU	<code>torch.nn.ReLU(inplace=False)</code> $\text{ReLU}(x) = \max(0, x)$	 <p>ReLU activation function</p>
ELU	<code>torch.nn.ELU(alpha=1.0, inplace=False)</code> $\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$	 <p>ELU activation function</p>
Hardshrink	<code>torch.nn.Hardshrink(lambd=0.5)</code> $\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$	 <p>Hardshrink activation function</p>

Hardtanh	<code>torch.nn.Hardtanh(min_val=-1.0, max_val=1.0, inplace=False, min_value=None, max_value=None)</code> $\text{HardTanh}(x) = \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ x & \text{otherwise} \end{cases}$	
LeakyReLU	<code>torch.nn.LeakyReLU(negative_slope=0.01, inplace=False)</code> $\text{LeakyReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative_slope} \times x, & \text{otherwise} \end{cases}$	
LogSigmoid	<code>torch.nn.LogSigmoid()</code> $\text{LogSigmoid}(x) = \log \left(\frac{1}{1 + \exp(-x)} \right)$	

Обучение нейронной сети

Если в качестве обучения используется градиентный спуск (алгоритм обратного распространения ошибки), то процесс обучения выполняется итерационно и включает прямой проход и обратный.

Прямой проход - вычисление выходов НС и текущей ошибки (функции потерь).

Для вычисления выходных значений НС необходимо, подать на входы НС данные из обучающей выборки и последовательно проходить все слои НС.

В библиотеке pytorch в класса Model за выполнение этого действия отвечает метод `forward`, именно в нем задаётся схема вычисления выходов НС.

Пример определения метода `forward` (см. выше определение НС):

```
def forward(self, input):
    x = self.pool1(F.relu(self.conv1(input)))
    x = self.pool2(F.relu(self.conv2(x)))
    x = x.view(x.size(0), -1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    return x
```

В PyTorch необязательно вызывать метод в явном виде, при создании объекта НС с указанием входного тензора, метод вызывается. Для вычисления ошибки (функции потерь), необходимо знать текущие значения выходов НС и ожидаемые (те, на которых обучаем). Существует несколько вариантов расчёта потерь (Таблица 4).

Таблица 4 – Примеры функций потерь (подробнее см. <https://pytorch.org/docs/stable/nn.html#loss-functions>)

Название функции	Синтаксис	Формула
<p>В зависимости от <code>reduction='none' 'mean' 'sum'</code></p> $\ell(x, y) = L = \{l_1, \dots, l_N\}^T,$ $\ell(x, y) = \begin{cases} \text{mean}(L), \\ \text{sum}(L), \end{cases}$ <p>Если для поля <code>size_average</code> установлено значение <code>False</code>, потери суммируются для каждой мини-партии. Игнорируется, когда уменьшить является ложным. По умолчанию: <code>True</code></p>		
L1Loss	<code>torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')</code>	$l_n = x_n - y_n ,$
MSELoss (средняя квадратичная)	<code>torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')</code>	$l_n = (x_n - y_n)^2$
KLDivLoss (информационного расхождения Кульбака-Лейблера)	<code>torch.nn.KLDivLoss(size_average=None, reduce=None, reduction='mean')</code>	$l_n = y_n \cdot (\log y_n - x_n)$
BCELoss (бинарной кросс-энтропии)	<code>torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')</code>	$l_n = -w_n [y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)],$
BCEWithLogitsLoss	<code>torch.nn.BCEWithLogitsLoss(weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None)</code>	$l_n = -w_n [y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))],$
HingeEmbeddingLoss	<code>torch.nn.HingeEmbeddingLoss(margin=1.0, size_average=None, reduce=None, reduction='mean')</code>	$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$

Пример определения функции потерь:

```
loss_fn = torch.nn.MSELoss(reduction='sum')
```

После того как определено, что НС ещё не обучена (количество итераций меньше заданного числа или функция потерь велика), необходимо обучить НС. Процесс обучения заключается в изменении параметров НС (весов), т.е. выполняется подбор оптимальных значений весов с учётом функции потерь и обучаемой выборки (заданных ожидаемых значений). Для этого используется метод градиентного спуска и реализуется обратный проход.

Веса можно изменять в ручную, изменяя Tensors, содержащие обучаемые параметры (например с помощью `torch.no_grad()` или `.data`). Это удобно в случае простых алгоритмов оптимизации, таких как стохастический градиентный спуск, но на практике мы часто обучаем нейронные сети, используя более сложные методы AdaGrad, RMSProp, Adam и т. д. Пакет `optim` в PyTorch абстрагирует идею алгоритма оптимизации и предоставляет реализации часто используемых алгоритмов оптимизации (Таблица 5).

Таблица 5 – Пример оптимизаторов (подробнее см. <https://pytorch.org/docs/stable/optim.html>)

Название	Метод
SGD	стохастический градиентный спуск
Adam	адаптивная оценка моментов
RMSprop	алгоритм Джеффри Хинтона
LBFGS	алгоритм Бroyдена-Флетчера-Гольдфарба-Шанно с ограниченным использованием памяти

Для использования оптимизатора необходимо на каждой итерации необходимо обнулять градиент, вызывать функцию `backward` и выполнять шаг оптимизатора.

Пример:

```
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Использование нейронной сети

После обучения НС ее требуется проверить на тестовой выборке. Если результат удовлетворителен, ее можно использовать для решения поставленной задачи, подавая на вход произвольные значения.

Примеры реализации нейронной сети

- 1) *Пример НС прямого распространения с функции активации ReLU, случайной выборкой, функцией потерь MSELoss, без использования оптимизатора, задана сеть с помощью nn.Sequential.*

Код:

```
# импорт библиотеки PyTorch
import torch
# задаем значения размерности
N, D_in, H, D_out = 64, 1000, 100, 10
# задаем случайным образом выборки
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
# строим модель НС
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
# выбрали функцию потерь
loss_fn = torch.nn.MSELoss(reduction='sum')
# скорость обучения
learning_rate = 1e-4
# цикл обучения с 500 эпохами
for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()
    loss.backward()
# расчет вручную параметрой НС
with torch.no_grad():
    for param in model.parameters():
        param -= learning_rate * param.grad
```

- 2) *Пример НС прямого распространения со случайной выборкой с 2 слоями, первый с функции активации ReLU, выходной с сигмоидальной функцией, используется для обучения метод обратного распространения и стохастический градиентный спуск (SGD), в качестве функции потерь средняя квадратичная функция (MSELoss).*

Код:

```
# импорт библиотеки PyTorch
import torch
import torch.nn as nn

# определить все слои и размер пакета
# n_in -входной, n_h - скрытый, n_out - выходной, batch_size - пакет
# обучающей выборки
n_in, n_h, n_out, batch_size = 10, 5, 1, 9
```

```

# заполняем случайными числами
#Возвращает тензор, заполненный случайными числами из нормального
распределения со средним 0 и дисперсией 1
#(также называемый стандартным нормальным распределением).Форма тензора
определяется переменным размером аргумента.

# входные данные
x = torch.randn(batch_size, n_in)
print(x)

#Создает тензор с данными.
#выходные данные
y = torch.tensor([[1.0], [0.0], [0.0], [1.0], [1.0], [1.0], [0.0], [0.0],
[1.0]])
print(y)

#class torch.nn.Sequential(*args) - Последовательный контейнер (модель НС).
#Модули будут добавлены к нему в порядке их передачи в конструктор.
# Линейное преобразование входных данных  $y=x*(A)^T+b$ 
# определяем слой входов 10, слой с функцией ReLU() содержит 5 нейронов,
выходной нейрон 1 с функцией сигмоида
model = nn.Sequential(nn.Linear(n_in, n_h),
    nn.ReLU(),
    nn.Linear(n_h, n_out),
    nn.Sigmoid())

# выбрали функцию потерь MSELoss()
criterion = torch.nn.MSELoss()
# выбрали метод оптимизации и установили скорость обучения
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)
# модель градиентного спуска
# цикл обучения
for epoch in range(100):
    # Прямой проход: вычисляем выход НС, подав на вход модели начальные
значения X
    y_pred = model(x)

    print(y_pred)
    # рассчитываем функцию потерь
    loss = criterion(y_pred, y)
    print('epoch: ', epoch, ' loss: ', loss.item())

    # Нулевые градиенты, выполнить обратный проход и обновить веса.
    # В PyTorch нам нужно установить градиенты на ноль, прежде чем начинать
обратное распространение,
    # поскольку PyTorch накапливает градиенты при последующих обратных
проходах.
    optimizer.zero_grad()

    # обратный проход (вычисляются градиенты)
    loss.backward()

    # шаг спуска градиента
    optimizer.step()

# новый входной вектор

```

```

x1 = torch.randn(1, n_in)
print(x1)

# получение выхода обученной НС
y_pred2 = model(x1)
print(y_pred2)

```

3) *Пример НС (Linear → ReLU → Linear), определённой с помощью nn.Module, с использованием оптимизатора Adam, с функцией потерь CrossEntropyLoss, обучающая выборка набор MNIST.*

Код:

```

import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
from torch.autograd import Variable
# Размеры изображения = 28 x 28 = 784
input_size = 784
# Количество узлов на скрытом слое
hidden_size = 500
# Число классов на выходе. В этом случае от 0 до 9
num_classes = 10
# Количество тренировок всего набора данных
num_epochs = 5
# Размер входных данных для одной итерации
batch_size = 100
# Скорость обучения
learning_rate = 0.001
# Грузим набор данных MNIST
# обучающая выборка
train_dataset = dsets.MNIST(
    root='./data',
    train=True,
    transform=transforms.ToTensor(),
    download=True
)
# тестовая выборка
test_dataset = dsets.MNIST(
    root='./data',
    train=False,
    transform=transforms.ToTensor()
)
# создаем загрузчик для НС с перемешиванием для обучающей выборки и без
перемешивания для тестовой.
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False
)

```



```

))
# Определяем вид НС
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__() # Наследуемый
родительским классом nn.Module
        self.fc1 = nn.Linear(input_size, hidden_size) # 1й связанный слой:
784 (данные входа) -> 500 (скрытый узел)
        self.relu = nn.ReLU() # Нелинейный слой ReLU
max(0,x)
        self.fc2 = nn.Linear(hidden_size, num_classes) # 2й связанный слой:
500 (скрытый узел) -> 10 (класс вывода)

    def forward(self, x): # Прямой проход
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

# создаем объект НС
net = Net(input_size, hidden_size, num_classes)
# Определяем функцию потерь
criterion = nn.CrossEntropyLoss()
# Определяем оптимизатор
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
# Цикл обучения
for epoch in range(num_epochs):
    # Грузим пакет изображений (index, data, class)
    for i, (images, labels) in enumerate(train_loader):
        # Преобразуем тензор в Variable: вектор 784 в матрицу 28 x 28
        images = Variable(images.view(-1, 28*28))
        labels = Variable(labels)
        # Инициализируем скрытые слои, веса=0
        optimizer.zero_grad()
        # Прямой проход по НС; вычисляем выход
        outputs = net(images)
        # Вычисляем функцию потерь (ошибку)
        loss = criterion(outputs, labels)
        # Обратный проход: корректировка весов
        loss.backward()
        # Выполняем шаг оптимизации (обучения)
        optimizer.step()
        # Выводим данные по обучению
        if (i+1) % 100 == 0:
            print('Epoch [%d/%d], Step [%d/%d], Loss: %.4f'%(epoch+1,
num_epochs, i+1, len(train_dataset)//batch_size, loss))
        correct = 0
        total = 0
    for images, labels in test_loader:
        images = Variable(images.view(-1, 28*28))
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1) # Выбор лучшего класса из
выходных данных: класс с лучшим счетом
        total += labels.size(0) # Увеличиваем суммарный счёт
        correct += (predicted == labels).sum() # Увеличиваем корректный счёт
    print('Точность сети на 10K тестовых изображений: %d %%' % (100 * correct /
total))

```

```
torch.save(net.state_dict(), 'fnn_model.pkl')
```

4) Пример Сверточной сети НС (Conv2d -> MaxPool2d -> Conv2d -> MaxPool2d -> Linear -> Linear), определённой с помощью nn.Module, с функцией потерь CrossEntropyLoss, случайной обучающей выборкой.

Код:

```
import torch
from torch.autograd import Variable
import torch.nn as nn
import torch.nn.functional as F
class MNISTConvNet(nn.Module):
    def __init__(self):
        super(MNISTConvNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(10, 20, 5)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
    def forward(self, input):
        x = self.pool1(F.relu(self.conv1(input)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        return x
net = MNISTConvNet()
print(net)
input = Variable(torch.randn(1, 1, 28, 28))
out = net(input)
print(out.size())
target = Variable(torch.LongTensor([3]))
loss_fn = nn.CrossEntropyLoss()
err = loss_fn(out, target)
err.backward()
print(err)
print(net.conv1.weight.data.norm())
print(net.conv1.weight.grad.data.norm())
```

5) Пример рекуррентной сети с функцией `relu` обучаемой на входных данных набора `MNIST`, оптимизатором `SGD` и функцией потерь `CrossEntropyLoss`.

[illegible]

```

        download=True)
test_dataset = dsets.MNIST(root='./data',
                            train=False,
                            transform=transforms.ToTensor())

# задаем параметры
batch_size = 100
n_iters = 3000
num_epochs = n_iters / (len(train_dataset) / batch_size)
num_epochs = int(num_epochs)
# подаем данные в загрузчик
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                           batch_size=batch_size,
                                           shuffle=False)

# определяем рекуррентную нейронную сеть
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()
        # размерность скрытых слоев]
        self.hidden_dim = hidden_dim
        # Количество скрытых слоев
        self.layer_dim = layer_dim
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True,
nonlinearity='relu')
        # Слой считывания
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(self.layer_dim, x.size(0),
self.hidden_dim).requires_grad_()
        out, hn = self.rnn(x, h0.detach())
        # out.size() --> 100, 28, 10
        out = self.fc(out[:, -1, :])
        # out.size() --> 100, 10
        return out

# задаем параметры НС
input_dim = 28
hidden_dim = 100
layer_dim = 1
output_dim = 10
learning_rate = 0.01
criterion = nn.CrossEntropyLoss()
model = RNNModel(input_dim, hidden_dim, layer_dim, output_dim)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
seq_dim = 28
iter = 0
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        model.train()
        # Загрузка изображений в виде тензоров с возможностью накопления
градиента
        images = images.view(-1, seq_dim, input_dim).requires_grad_()
        # Обнуление градиента
        optimizer.zero_grad()

```

```

# Прямой проход
outputs = model(images)
# Рассчитываем функцию потерь
loss = criterion(outputs, labels)
# Обратный проход
loss.backward()
# Шаг градиентного спуска (изменение параметров НС)
optimizer.step()
iter += 1
if iter % 500 == 0:
    #производим оценки НС
    model.eval()
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.view(-1, seq_dim, input_dim)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum()
    accuracy = 100 * correct / total
    print('Итерация: {}'. Потери: {}. Точность: {}'.format(iter,
loss.item(), accuracy))

```

Варианты заданий

Вариант	Функция потерь	Функция активации	НС	Оптимизатор	Входные данные
1	L1Loss	Sigmoid	Свёрточная НС (не менее 2 свёрточных слоёв и одного линейного)	RMSProp	MNIST
2	MSELoss	ReLU	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	SGD	CIFAR10
3	KLDivLoss	ELU	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	Adam	STL10
4	BCELoss	Hardshrink	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	Adagrad	Рандом
5	BCEWithLogitsLoss	Hardtanh	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	SGD	CIFAR10
6	HingeEmbeddingLoss	LeakyReLU	Свёрточная НС (не менее 2 свёрточных слоёв и одного линейного)	Adam	STL10
7	KLDivLoss	LogSigmoid	Свёрточная НС (не менее 2 свёрточных слоёв и одного линейного)	RMSProp	CIFAR10
8	BCELoss	ELU	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	SGD	STL10
9	BCEWithLogitsLoss	Hardshrink	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	Adam	MNIST
10	L1Loss	Hardtanh	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	RMSProp	CIFAR10
11	MSELoss	LeakyReLU	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	SGD	STL10
12	KLDivLoss	LogSigmoid	Свёрточная НС (не менее 2 свёрточных слоёв и одного линейного)	Adam	Рандом
13	BCELoss	Sigmoid	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	Adagrad	CIFAR10
14	KLDivLoss	LogSigmoid	Свёрточная НС (не менее 2 свёрточных слоёв и одного линейного)	RMSProp	CIFAR10
15	BCELoss	ELU	Рекуррентная нейронная сеть с 2 нелинейными слоями (readout)	SGD	STL10
16	BCEWithLogitsLoss	Hardshrink	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	Adam	CIFAR10
17		Hardtanh	НС прямого распространения (использовать не менее 2 разных нелинейных функций)	RMSProp	STL10

*Если параметры варианта не совместимы, обоснуйте и измените.