Искусственные нейронные сети: основы практического применения

Лекция 6

Крощенко А.А.

Брестский государственный технический университет

07.06.2017

Рекуррентные нейронные сети

- **Рекуррентными** называются такие сети, в которых выходы нейронных элементов **последующих** слоев соединены с нейронами **предшествующих** слоев
- Учет результатов преобразований информации НС на предыдущем этапе (-ах) позволяет использовать такие сети естественным образом для решения задач **прогнозирования** и **управления**
- К простейшим типам РНС относятся нейронные сети **Джоржана, Элмана и мультирекуррентная НС**. Эти сети являются двухслойными (два обрабатывающих слоя). При этом на скрытом слое часто устанавливается функция активации гиперболический тангенс или сигмоида. На выходном слое – линейная функция активации. Рассмотрим их подробнее

Простейшие рекуррентные нейронные сети: сеть Джордана

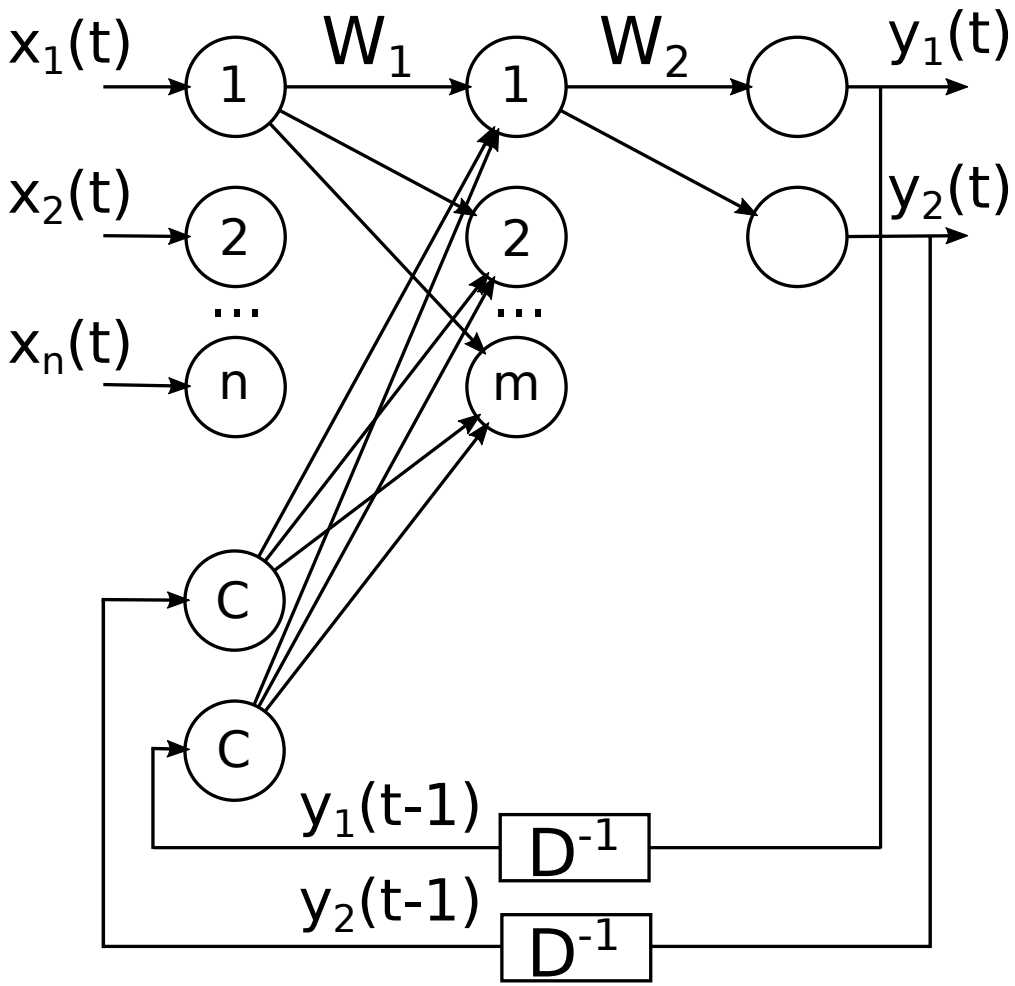


Figure 1: Схема сети Джордана, C – контекстные нейроны

Простейшие рекуррентные нейронные сети: сеть Элмана

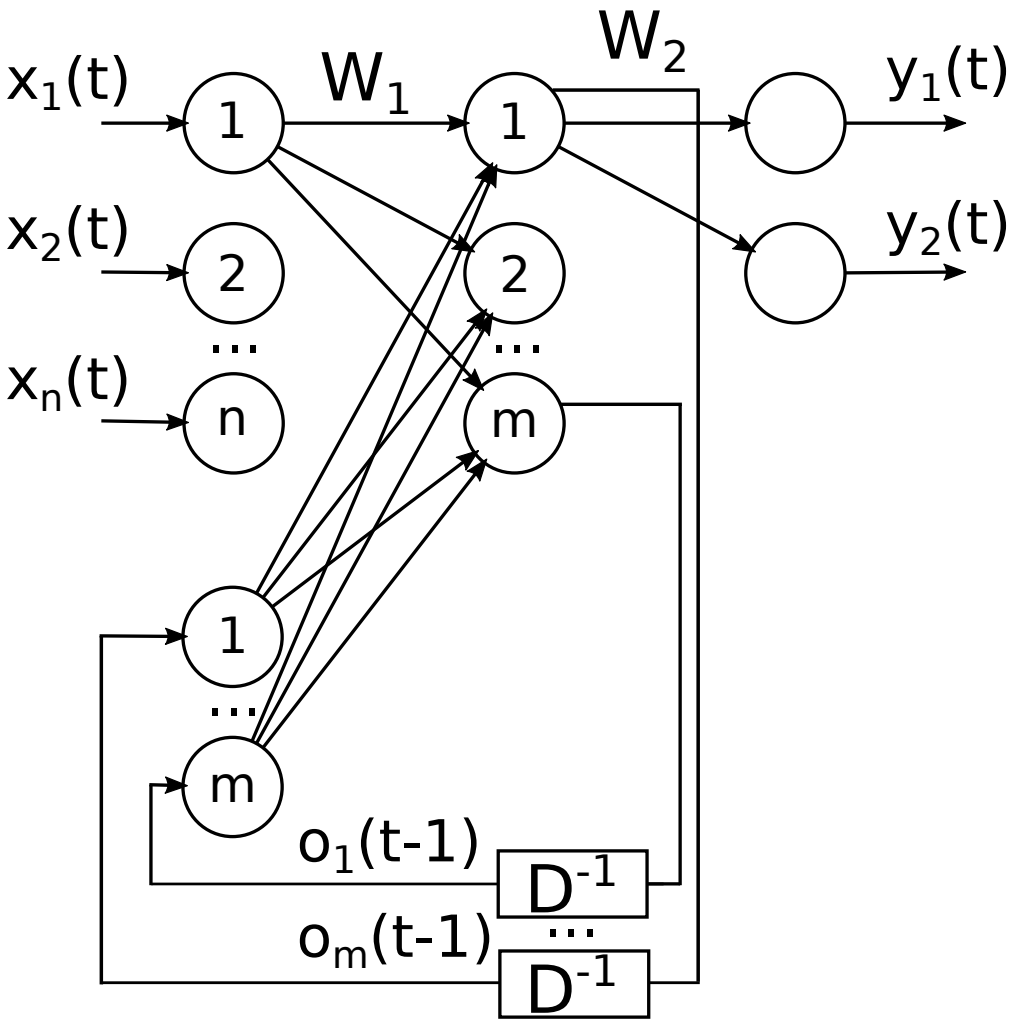


Figure 2: Схема сети Элмана

Мультирекуррентная сеть

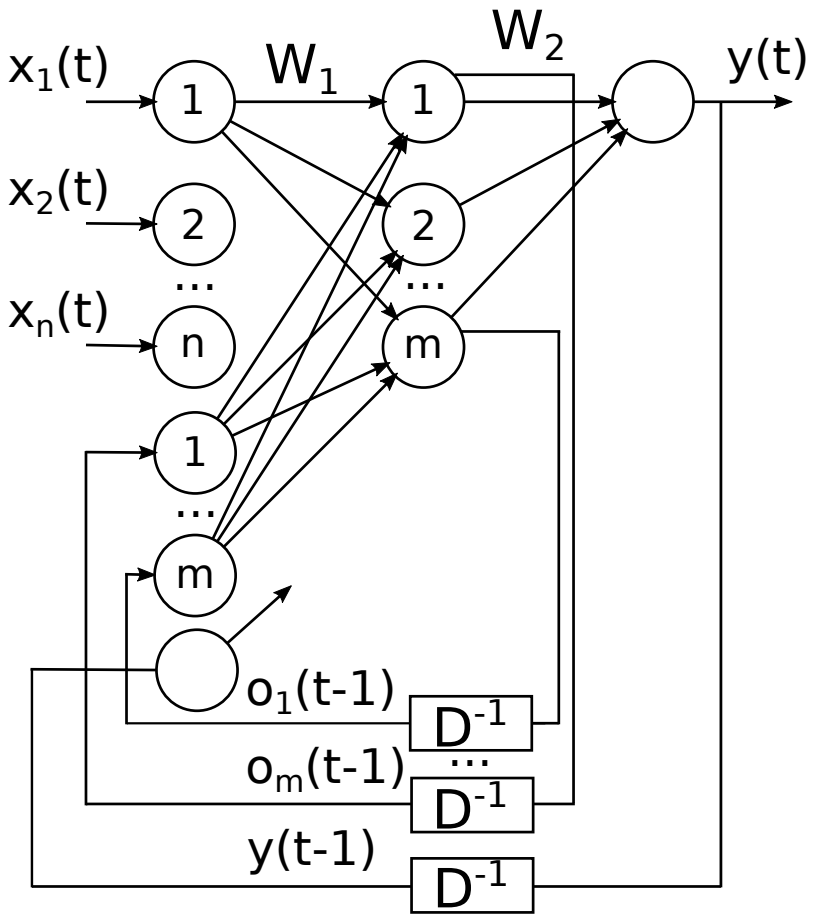


Figure 3: Схема мультирекуррентной сети

Обучение рекуррентных нейронных сетей

Рекуррентная нейронная сеть в общем случае обучается с помощью алгоритма обратного распространения ошибки с учетом наличия контекстных нейронов.
Алгоритм обучения будет иметь следующий вид:

1. В начальный момент времени $t = 1$ все контекстные нейроны устанавливаются в нулевое состояние, т.е. их выходные значения равняются нулю, т.е., например, для сети Джордана $y_i(0) = 0$.
2. Входной образ подается на сеть и происходит прямое распространение его в нейронной сети
3. В соответствии с алгоритмом обратного распространения ошибки производится модификация весовых коэффициентов и пороговых значений нейронных элементов
4. t = t + 1 и осуществляется переход к пункту 2

Обучение производится до выполнения условия останова (суммарная ошибка станет меньше предопределенной или будет превышено максимальное число эпох обучения).

Реализация: нейронная сеть Джордана

```
from activate_functions import Logistic
import itertools as it

class JordanNetwork:
  def __init__(self, inputs, hidden, outputs, max_epochs, min_error
      , rate):
    self.weights_hidden=0.2*np.random.random((inputs,hidden))-0.1
    self.weights_outputs=0.2*np.random.random((hidden,outputs))-0.1
    self.recurrent_weights=0.2*np.random.random((outputs,hidden))
        -0.1
    self.biases_hidden=0.2*np.random.random((1,hidden))-0.1
    self.biases_output=0.2*np.random.random((1,outputs))-0.1
    self.max_epochs = max_epochs
    self.min_error = min_error
    self.rate = rate
    self.inputs = inputs
    self.context = []
    self.act_hidden = Logistic()
```

```
def train(self, data, targets):
  epochs_count = 0; isFinish = False
  while not isFinish:
    i = 0; error = 0
    for sample, target in it.izip(data, targets):
      activate_hidden = np.dot(sample, self.weights_hidden) + self.
          biases_hidden
      if i + epochs_count == 0:
        activate_hidden = self.act_hidden.apply(activate_hidden)
      else:
        activate_hidden = self.act_hidden.apply(activate_hidden +
            np.dot(self.context, self.recurrent_weights))
      activate_output = np.dot(activate_hidden, self.
          weights_outputs) + self.biases_output
      output_error = activate_output - target
      hidden_error = np.dot(output_error, self.weights_outputs.T) *
           activate_hidden * (1 - activate_hidden)
      change_recurrent_weights = i + epochs_count > 0
      self.change_weights(sample, output_error, activate_hidden,
          hidden_error, change_recurrent_weights)
      self.context = activate_output
      i += 1
      error += (output_error * output_error).sum()
    epochs_count += 1
    isFinish = epochs_count > self.max_epochs or error < self.
        min_error
```

```
def change_weights(self, sample, output_error, activate_hidden,
    hidden_error, change_recurrent_weights):
  self.weights_outputs -= self.rate * np.dot(activate_hidden.T,
      output_error)
  self.biases_output -= self.rate * output_error.sum(axis=0)
  self.weights_hidden -= self.rate * np.dot(sample.reshape((len(
      sample), 1)), hidden_error)
  self.biases_hidden -= self.rate * hidden_error.sum(axis=0)
  if change_recurrent_weights:
    self.recurrent_weights -= self.rate * np.dot(self.context.T,
        hidden_error)

def activate(self, sample):
  activate_hidden = self.act_hidden.apply(
  np.dot(sample, self.weights_hidden) + np.dot(self.context, self.
      recurrent_weights) + self.biases_hidden)
  activate_output = np.dot(activate_hidden, self.weights_outputs) +
       self.biases_output
  self.context = activate_output
  return activate_output
```

Реализация: нейронная сеть Элмана

```
class ElmanNetwork:
    def __init__(self, inputs, hidden, outputs, max_epochs,
        min_error, rate):
        self.weights_hidden=0.2*np.random.random((inputs,hidden))-0.1
        self.weights_outputs=0.2*np.random.random((hidden,outputs))
            -0.1
        self.recurrent_weights=0.2*np.random.random((hidden,hidden))
            -0.1
        self.biases_hidden=0.2*np.random.random((1,hidden))-0.1
        self.biases_output=0.2*np.random.random((1,outputs))-0.1
        self.max_epochs = max_epochs
        self.min_error = min_error
        self.rate = rate
        self.inputs = inputs
        self.context = []
        self.act_hidden = Logistic()
```

```
def train(self, data, targets):
    epochs_count = 0; isFinish = False
    while not isFinish:
        i = 0; error = 0
        for sample, target in it.izip(data, targets):
            activate_hidden=np.dot(sample,self.weights_hidden)+self.
                biases_hidden
            if i + epochs_count == 0:
                activate_hidden=self.act_hidden.apply(activate_hidden)
            else:
                activate_hidden = self.act_hidden.apply(
                activate_hidden + np.dot(self.context, self.
                    recurrent_weights))
            activate_output = np.dot(activate_hidden, self.
                weights_outputs) + self.biases_output
            output_error = activate_output - target
            hidden_error = np.dot(output_error, self.weights_outputs.T
                ) * activate_hidden * (1 - activate_hidden)
            change_recurrent_weights = i + epochs_count > 0
            self.change_weights(sample, output_error, activate_hidden,
                 hidden_error, change_recurrent_weights)
            self.context = activate_hidden
        i += 1
        error+=(output_error*output_error).sum(); epochs_count+=1
        isFinish = epochs_count > self.max_epochs or error < self.
            min_error
```

```
def change_weights(self, sample, output_error, activate_hidden,
    hidden_error, change_recurrent_weights):
    self.weights_outputs -= self.rate * np.dot(activate_hidden.T,
        output_error)
    self.biases_output -= self.rate * output_error.sum(axis=0)
    self.weights_hidden -= self.rate * np.dot(sample.reshape((len(
        sample), 1)), hidden_error)
    self.biases_hidden -= self.rate * hidden_error.sum(axis=0)
    if change_recurrent_weights:
        self.recurrent_weights -= self.rate * np.dot(self.context.T,
            hidden_error)

def activate(self, sample):
    activate_hidden = self.act_hidden.apply(
    np.dot(sample, self.weights_hidden) + np.dot(self.context, self.
        recurrent_weights) + self.biases_hidden)
    self.context = activate_hidden
    activate_output = np.dot(activate_hidden, self.weights_outputs)
        + self.biases_output
    return activate_output
```

Реализация: мультирекуррентная нейронная сеть

```
class MultiRecurrentNetwork:
    def __init__(self, inputs, hidden, outputs, max_epochs,
        min_error, rate):
        self.weights_hidden = 0.2 * np.random.random((inputs, hidden)
            ) - 0.1
        self.weights_outputs = 0.2 * np.random.random((hidden,
            outputs)) - 0.1
        self.recurrent_weights = 0.2 * np.random.random((hidden,
            hidden)) - 0.1
        self.recurrent_weights_output = 0.2 * np.random.random((
            outputs, hidden)) - 0.1
        self.biases_hidden = 0.2 * np.random.random((1, hidden)) -
            0.1
        self.biases_output = 0.2 * np.random.random((1, outputs)) -
            0.1
        self.context = []
        self.context_output = []
        self.max_epochs = max_epochs
        self.min_error = min_error
        self.rate = rate
        self.act_func = Logistic()
```

```
def train(self, data, targets):
    epochs_count = 0; isFinish = False
    while not isFinish:
        i = 0; error = 0
        for sample, target in it.izip(data, targets):
            activate_hidden = np.dot(sample, self.weights_hidden) +
                self.biases_hidden
            if i + epochs_count == 0:
                activate_hidden = self.act_func.apply(activate_hidden)
            else:
                activate_hidden = self.act_func.apply(
                activate_hidden + np.dot(self.context_output, self.
                    recurrent_weights_output) +
                np.dot(self.context, self.recurrent_weights))
```

```
    activate_output = np.dot(activate_hidden, self.
        weights_outputs) + self.biases_output
    output_error = activate_output - target
    hidden_error = np.dot(output_error, self.weights_outputs.T) *
         activate_hidden * (1 - activate_hidden)
    change_recurrent_weights = i + epochs_count > 0
    self.change_weights(sample, output_error, activate_hidden,
        hidden_error, change_recurrent_weights)
    self.context = activate_hidden
    self.context_output = activate_output
i += 1
error += (output_error * output_error).sum(); epochs_count += 1
isFinish = epochs_count > self.max_epochs or error < self.
    min_error
```

```
def change_weights(self, sample, output_error, activate_hidden,
    hidden_error, change_recurrent_weights):
   self.weights_outputs -= self.rate * np.dot(activate_hidden.T,
       output_error)
   self.biases_output -= self.rate * output_error.sum(axis=0)
   self.weights_hidden -= self.rate * np.dot(sample.reshape((len(
       sample), 1)), hidden_error)
   self.biases_hidden -= self.rate * hidden_error.sum(axis=0)
   if change_recurrent_weights:
      self.recurrent_weights -= self.rate * np.dot(self.context.T,
          hidden_error)
      self.recurrent_weights_output -= self.rate * np.dot(self.
          context_output.T, hidden_error)

def activate(self, sample):
  activate_hidden = self.act_func.apply(
    np.dot(sample, self.weights_hidden) + np.dot(self.
        context_output, self.recurrent_weights_output) +
    np.dot(self.context, self.recurrent_weights) + self.
        biases_hidden)
  self.context = activate_hidden
  activate_output = np.dot(activate_hidden, self.weights_outputs) +
      self.biases_output
  self.context_output = activate_output
  return activate_output
```

Применение рекуррентных нейронных сетей к решению задачи прогнозирования

Применим введенные рекуррентные нейронные сети для решения задачи прогнозирования временного ряда, генерируемого

1. функцией $\sin(x)$
2. нелинейной системой Лоренца

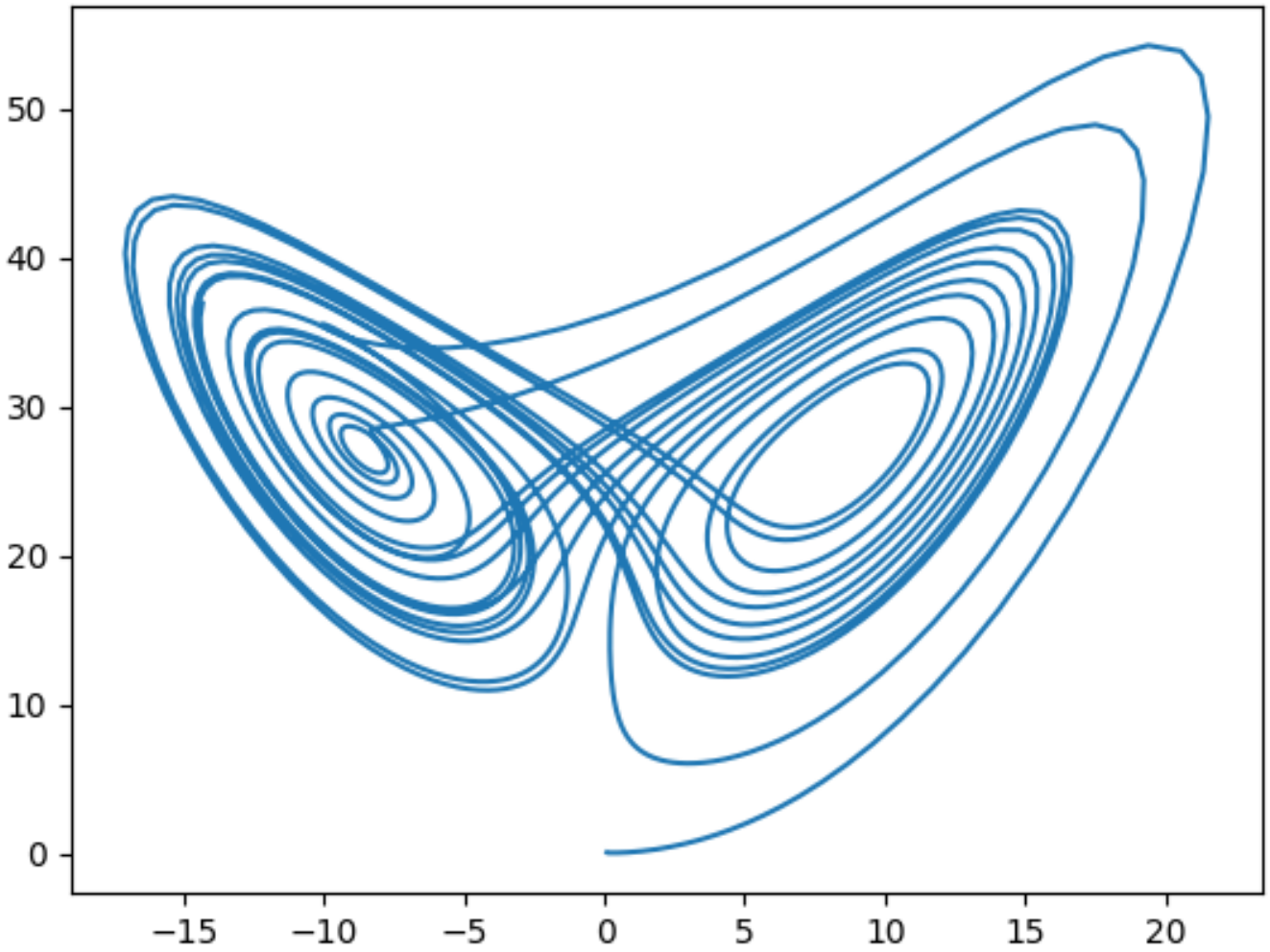
$$\begin{cases} \dot{x} = \sigma(y - x) \\ \dot{y} = x(r - z) - y \\ \dot{z} = xy - bz \end{cases}$$

Описание эксперимента

Для решения поставленной задачи применялись все основные типы простейших рекуррентных сетей. Во всех случаях использовалась архитектура: 7-14{Logistic}-1{Linear}.
Другие параметры обучения:

- Количество эпох – 2000 (система Лоренца – 100)
- Скорость обучения – 0.005
- Минимальная ошибка – 1e-5

Визуализация системы Лоренца



Формирование датасета

```
def form_dataset(data, window):
  i = 0
  dataset = []
  labels = []
  while i + window < len(data):
    dataset.append(data[i:i+window])
    labels.append(data[i+window])
    i += 1
  return np.array(dataset), np.array(labels)
```

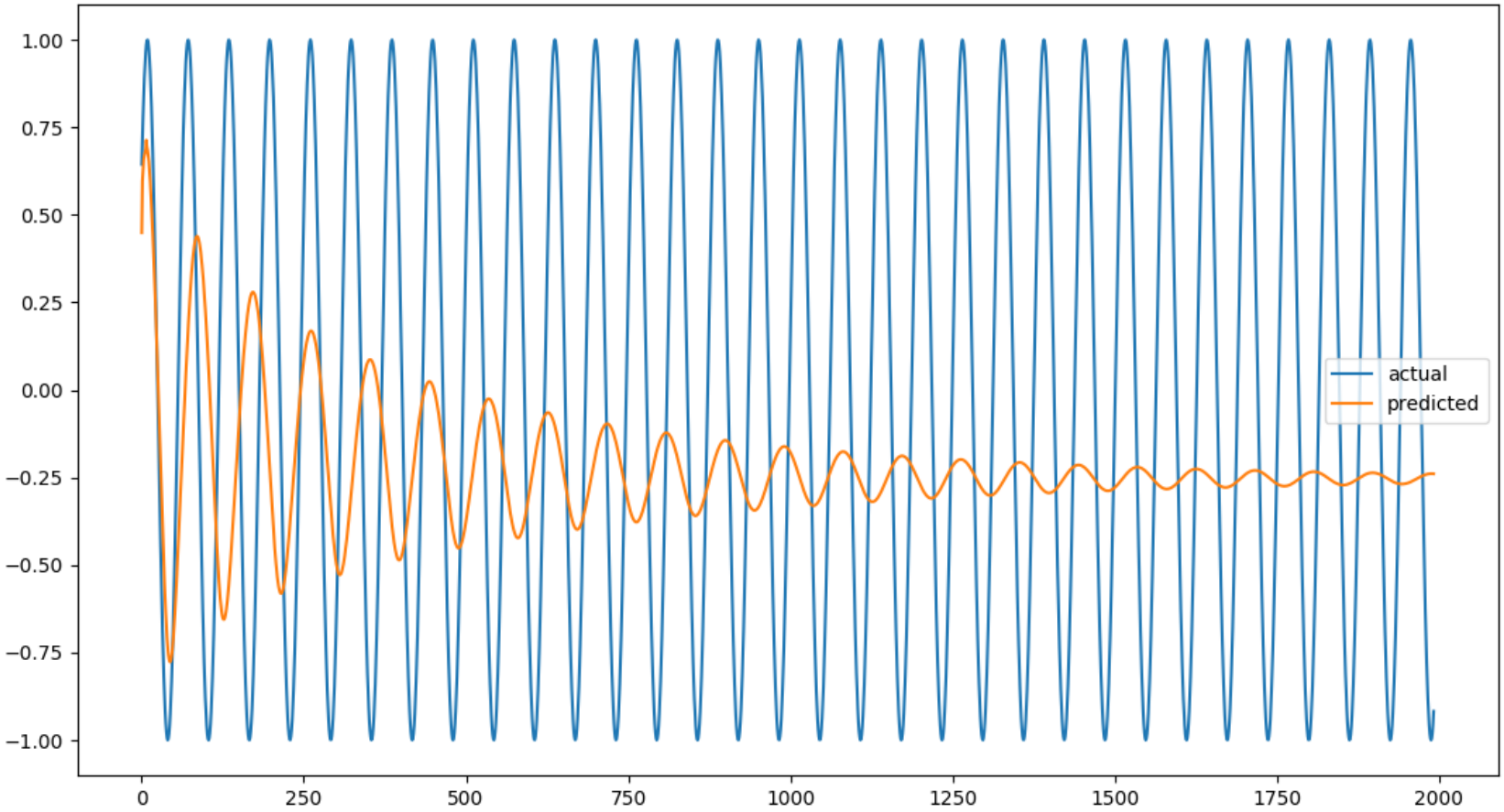
Как прогнозировать? Первый способ

```
def prediction_visualise(data, net, wondow):
    plt.plot(range(0, len(data)-window), data[window:])
    dataset, labels = form_dataset(data, window)
    i = 0
    predicted = []
    sample = dataset[0]
    while i < len(data) - window:
        predicted_value = net.activate(sample)
        predicted.append(predicted_value)
        sample[0:6] = sample[1:]
        sample[6] = predicted_value
        i += 1
    predicted = np.array(predicted)
    plt.plot(range(0, len(predicted)), predicted.reshape((len(
        predicted), 1)))
    plt.show()
```

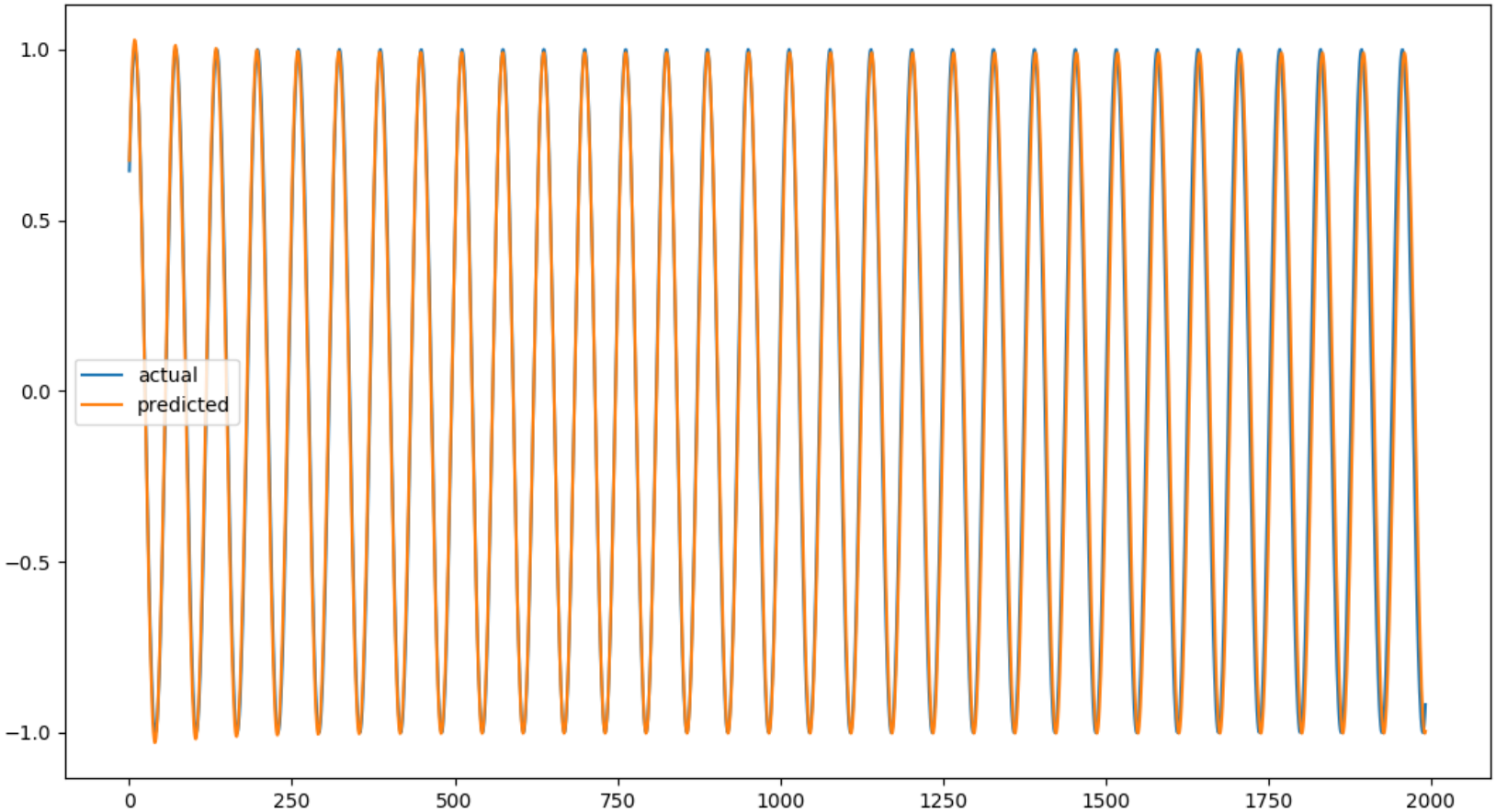
Как прогнозировать? Второй способ

```
def prediction_visualise(data, net, window):
    plt.plot(range(0, len(data)-window), data[window:])
    dataset, labels = form_dataset(data, window)
    dataset_nrm=(dataset-dataset.min())/(dataset.max()-dataset.min())
    i = 0
    predicted = []
    for sample in dataset_nrm:
        predicted.append(net.activate(sample))
            i += 1
    predicted = np.array(predicted)
    plt.plot(range(0, len(predicted)), predicted.reshape((len(
        predicted), 1)))
    plt.show()
```

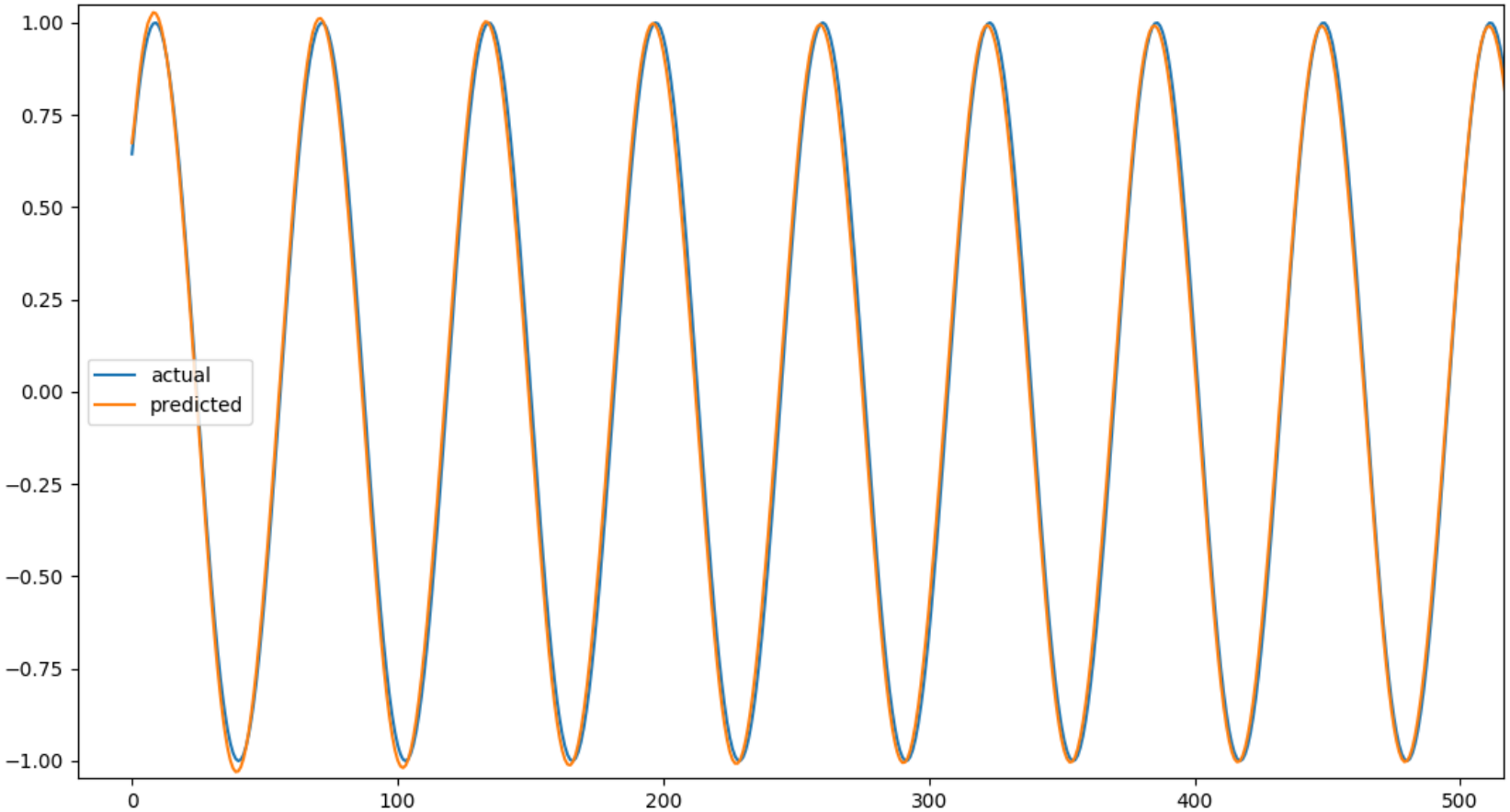
Результаты. sin(x), визуализация после 10 эпох обучения



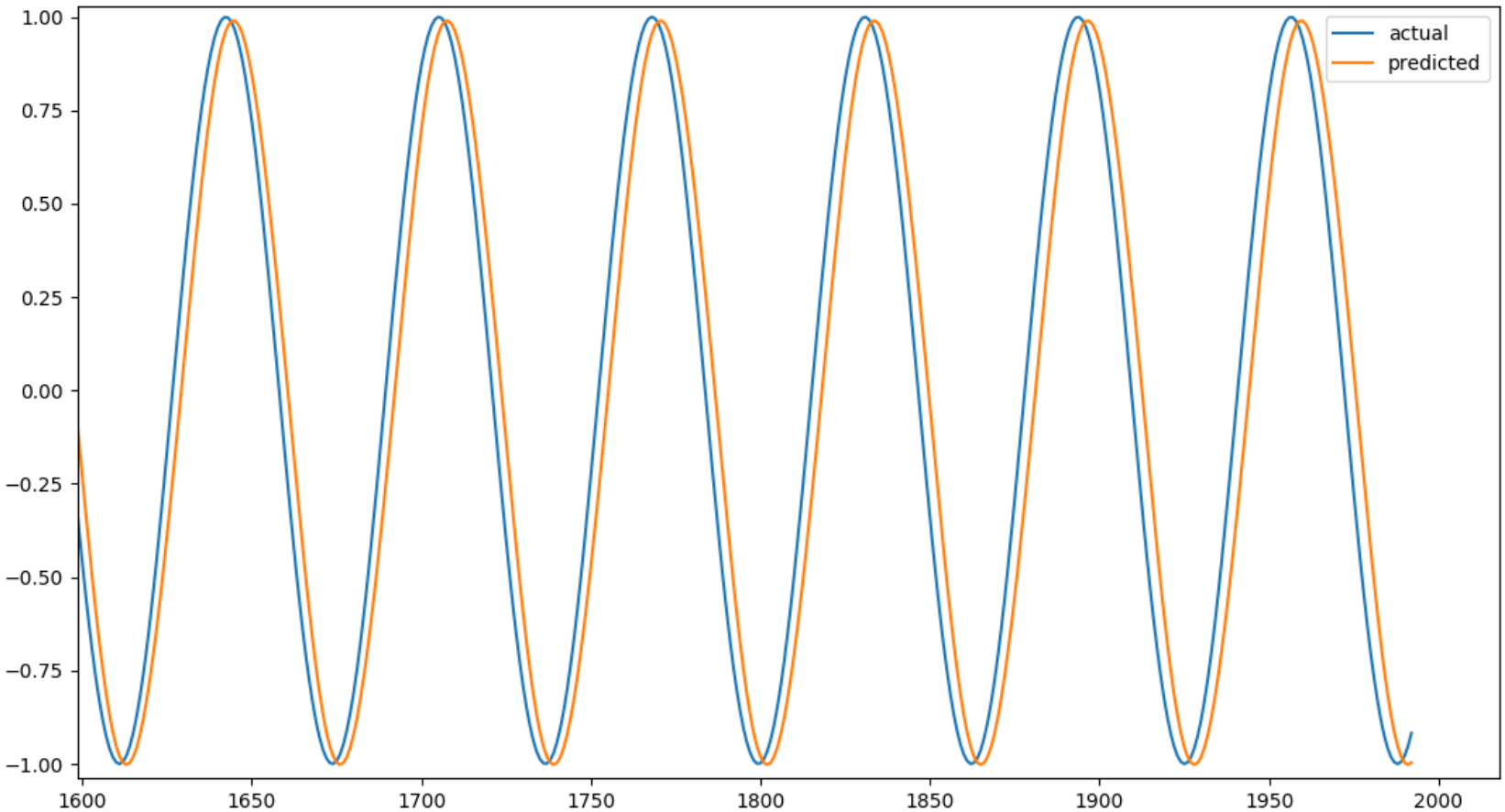
Результаты. sin(x), визуализация после 2000 эпох



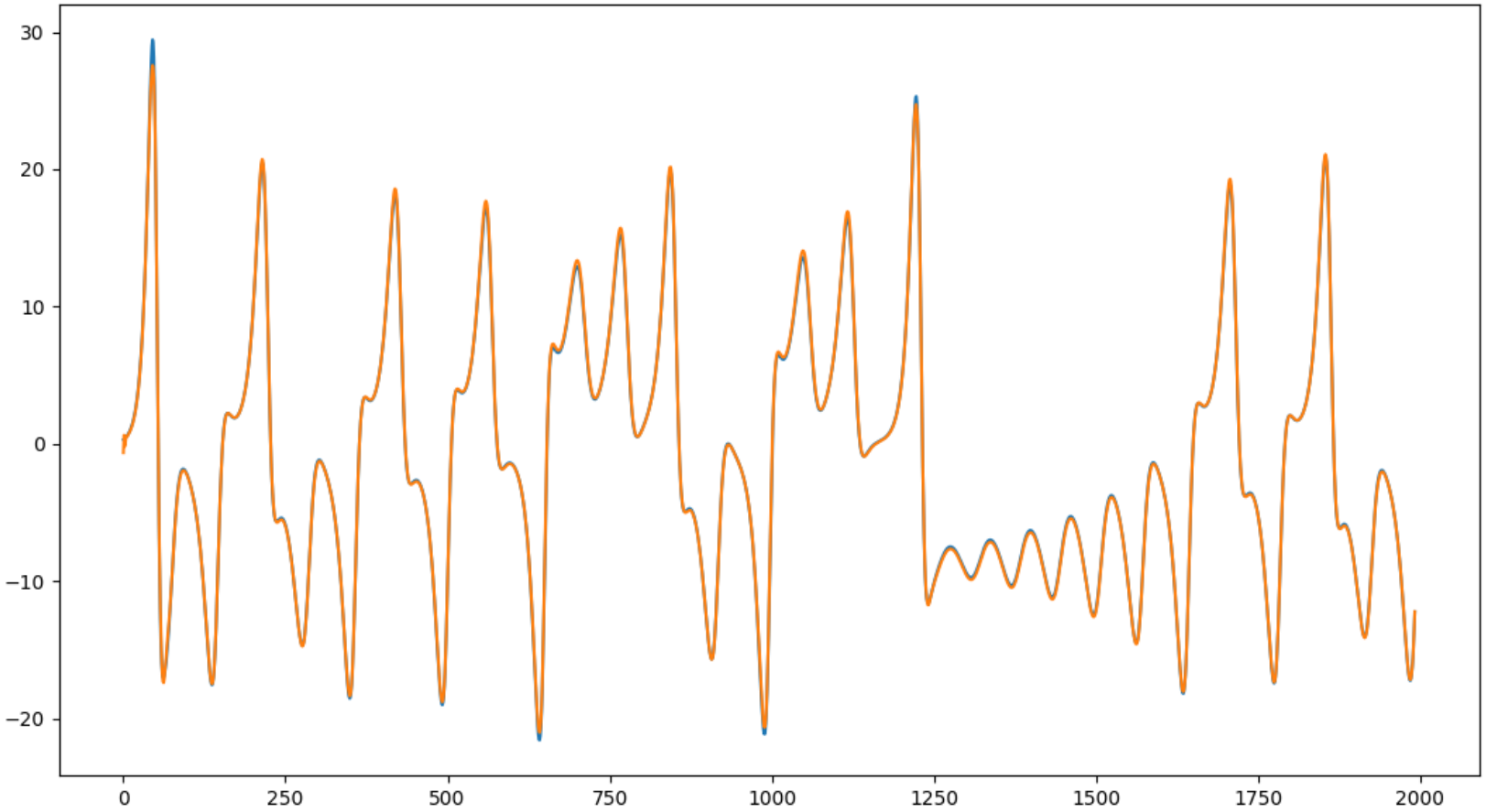
Результаты. sin(x), после 2000 эпох, начало



Результаты. sin(x), после 2000 эпох, конец



Результаты. Система Лоренца, после 100 эпох, Элман

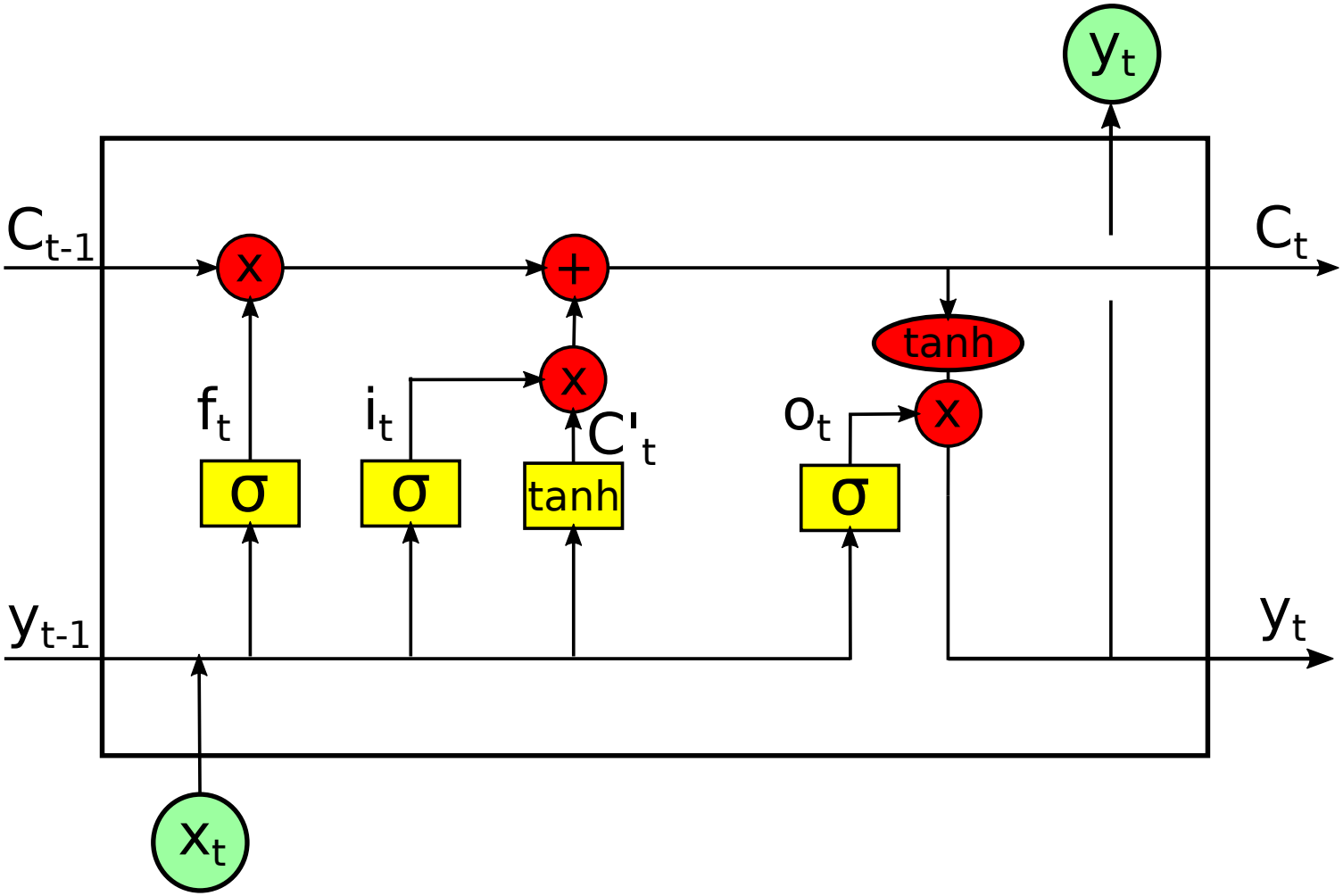


LSTM: Long short-term memory

- Представляет собой разновидность рекуррентной НС
- Применяется для решения задач прогнозирования, обработки естественной речи, классификации и т.д.
- Ключевой особенностью является способность работать с интересующими данными, разделенными **неопределенными задержками** во времени
- Имеет модульную структуру

Популярно о LSTM:
`http://alexsosn.github.io/ml/2015/11/16/LSTM.html`

Структура LSTM



Основные формулы, обучение LSTM

$$f_t = \sigma(W_f x_t + U_f y_{t-1} + b_f)$$

$$i_t = \sigma(W_i x_t + U_i y_{t-1} + b_i)$$

$$C'_t = \tanh(W_C x_t + U_C y_{t-1} + b_C)$$

$$C_t = f_t \circ C_{t-1} + i_t \circ C'_t$$

$$o_t = \sigma(W_o x_t + U_o y_{t-1} + b_o)$$

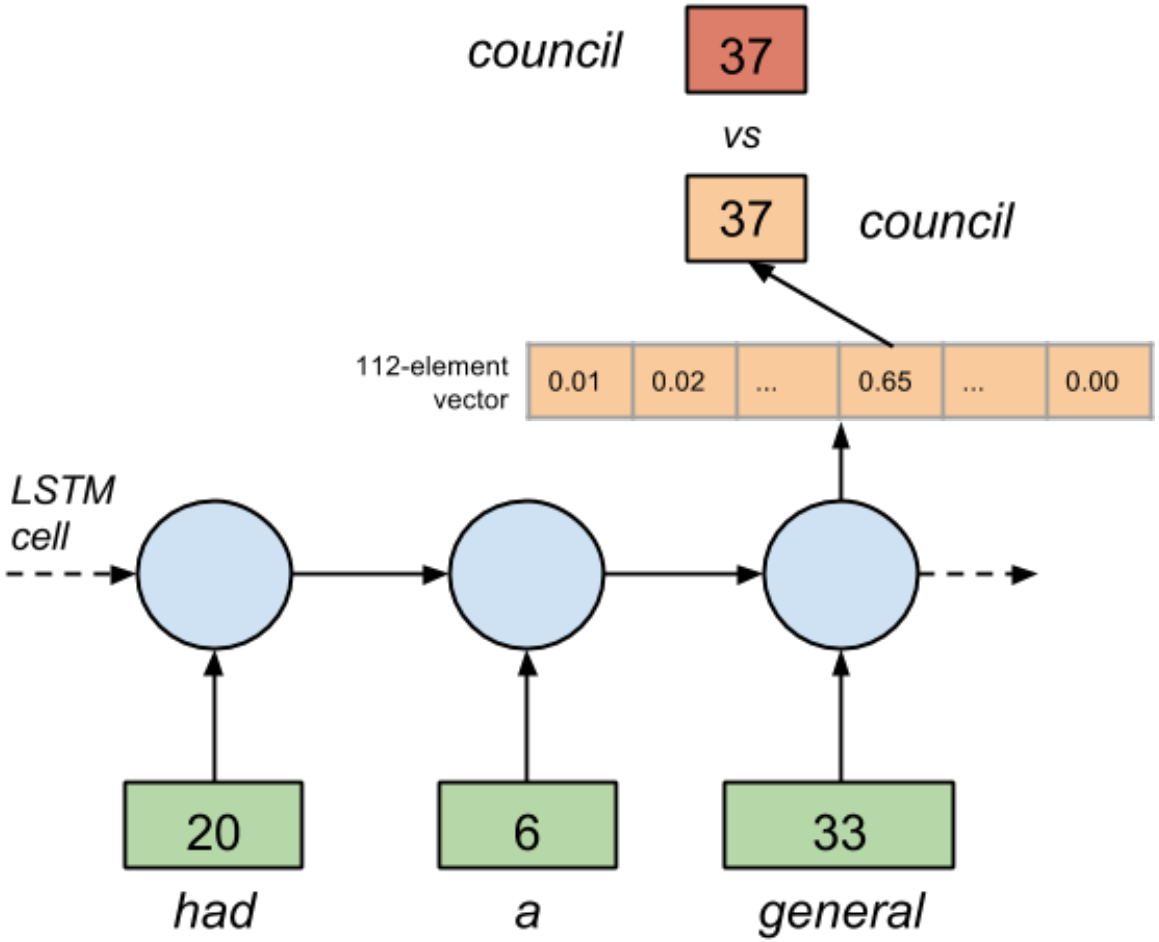
$$y_t = o_t \circ \tanh(C_t)$$

f_t называется вектором **вентиля «забывания»** (forget gate vector), i_t – вектором **входного вентиля** (input gate vector), а o_t – вектором **выходного вентиля** (output gate vector).
Обучение LSTM производится в соответствии с адаптированным алгоритмом обратного распространения ошибки

Задача

посадил дед репку . выросла репка сладка , крепка , большая - пребольшая . пошел дед репку рвать : тянет - потянет , вытянуть не может . позвал дед бабку . бабка за дедку , дедка за репку - тянут - потянут , вытянуть не могут . позвала бабка внучку . внучка за бабку , бабка за дедку , дедка за репку - тянут - потянут , вытянуть не могут . позвала внучка жучку . жучка за внучку , внучка за бабку , бабка за дедку , дедка за репку - тянут - потянут , вытянуть не могут . позвала жучка кошку . кошка за жучку , жучка за внучку , внучка за бабку , бабка за дедку , дедка за репку - тянут - потянут , вытянуть не могут . позвала кошка мышку . мышка за кошку , кошка за жучку , жучка за внучку , внучка за бабку , бабка за дедку , дедка за репку - тянут - потянут - и вытянули репку .

Идея решения



https://medium.com/towards-data-science/lstm-by-example-using-tensorflow-feb0c1968537

Результат обучения сети LSTM

- Запрос к сети: 'посадил дед репку' Ответ сети: 'посадил дед репку . выросла репка сладка , крепка , большая - пребольшая . пошел дед репку рвать : тянет - крепка , вытянуть не могут . позвала за бабку , дедка за репку -'
- Запрос к сети: 'кошка за жучку' Ответ сети: 'кошка за жучку , жучка за внучку , внучка за бабку , дедка за репку - тянут - потянут , вытянуть не могут . позвала за бабку , дедка за репку - тянут - потянут'
- Запрос к сети: 'бабка за дедку' Ответ сети: 'бабка за дедку , дедка за репку - тянут - потянут , вытянуть не могут . позвала за бабку , дедка за репку - тянут - потянут , вытянуть не могут . позвала за бабку'
- Запрос к сети: 'дедка бабка внучка' Ответ сети: 'дедка бабка внучка - внучка за и вытянули репку . выросла репка сладка , крепка , большая - пребольшая . пошел дед репку рвать : тянет - крепка , вытянуть не могут . позвала за'

Фреймворк Tensorflow: обзор основных возможностей

Библиотека Tensorflow – открытая библиотека для машинного обучения, разработанная компанией Google. Работа с библиотекой организуется в рамках т.н. **сессий**. До входа в сессию происходит настройка используемой модели, определение параметров алгоритма обучения, минимизируемой функции и т.д.
После начала сессии все ранее прописанные установки начинают действовать. Таким образом, Tensorflow «откладывает» вычисления до начала сессии.
Ключевым аспектом работы с Tensorflow является **граф**. Граф – своеобразное средство управления вычислениями на Tensorflow. При создании и формировании графа создается и формируется последовательность выполняемых операций.
Такая логика построения фреймворка позволяет выполнять дорогостоящие операции **сразу, пакетно**, тем самым ускоряя работу программы.

Алгоритмы искусственных иммунных систем и нейронных сетей для обнаружения вредоносных программ

Автор: к.т.н. Безобразов С.В.

Несмотря на активные действия со стороны производителей антивирусного программного обеспечения, компьютерные вирусы продолжают успешно проникать в компьютерные системы пользователей по всему миру и выполнять вредоносные действия по уничтожению или краже информации. Имеющиеся антивирусные программы не способны обеспечить надежную защиту. Применение ИНС способно обеспечить обнаружение даже **неизвестных** вредоносных программ.

Архитектура системы

Система базируется на использовании т.н. **иммунных детекторов**, каждый из которых представляет собой нейронную сеть.

Выходные значения детектора формируются после подачи всех образов на него в соответствии с выражением:

$$Z_1 = \begin{cases} 1, \text{если файл чистый} \\ 0, \text{если присутствует вирус} \end{cases}$$

$$Z_2 = \begin{cases} 1, \text{если файл заражен} \\ 0, \text{иначе} \end{cases}$$

Обучающая выборка

Обучающая выборка формируется из чистых файлов (класс чистых программ) и вредоносных программ (класс вредоносных программ). Присутствие вируса или его сигнатуры при обучении позволяет обученным иммунным детекторам находить разницу между чистыми файлами и компьютерными вирусами.
Желательно также иметь представителей всех типов вредоносных программ – черви, троянские программы, макровирусы и т.д.

$$X_i = \begin{bmatrix} X_i^1 \\ X_i^2 \\ \dots \\ X_i^L \end{bmatrix} = \begin{bmatrix} X_{i1}^1 & X_{i2}^1 & \dots & X_{in}^1 \\ X_{i1}^2 & X_{i2}^2 & \dots & X_{in}^2 \\ \dots & & & \\ X_{i1}^L & X_{i2}^L & \dots & X_{in}^L \end{bmatrix}$$

где L – размерность обучающей выборки. Соответственно, множество эталонов выглядит так

$$X_i = \begin{bmatrix} I_i^1 \\ I_i^2 \\ \dots \\ I_i^L \end{bmatrix} = \begin{bmatrix} I_{i1}^1 & I_{i2}^1 \\ I_{i1}^2 & I_{i2}^2 \\ \dots & \\ I_{i1}^L & I_{i2}^L \end{bmatrix}$$

Детали обучения детекторов

Нейронная сеть обучается путем **обучения с учителем**.
Целью обучения является минимизация ошибки

$$E_i = \frac{1}{2}\sum_{k=1}^{L}\sum_{j=1}^{2}(Z_{ij}^k - I_{ij}^k)^2$$

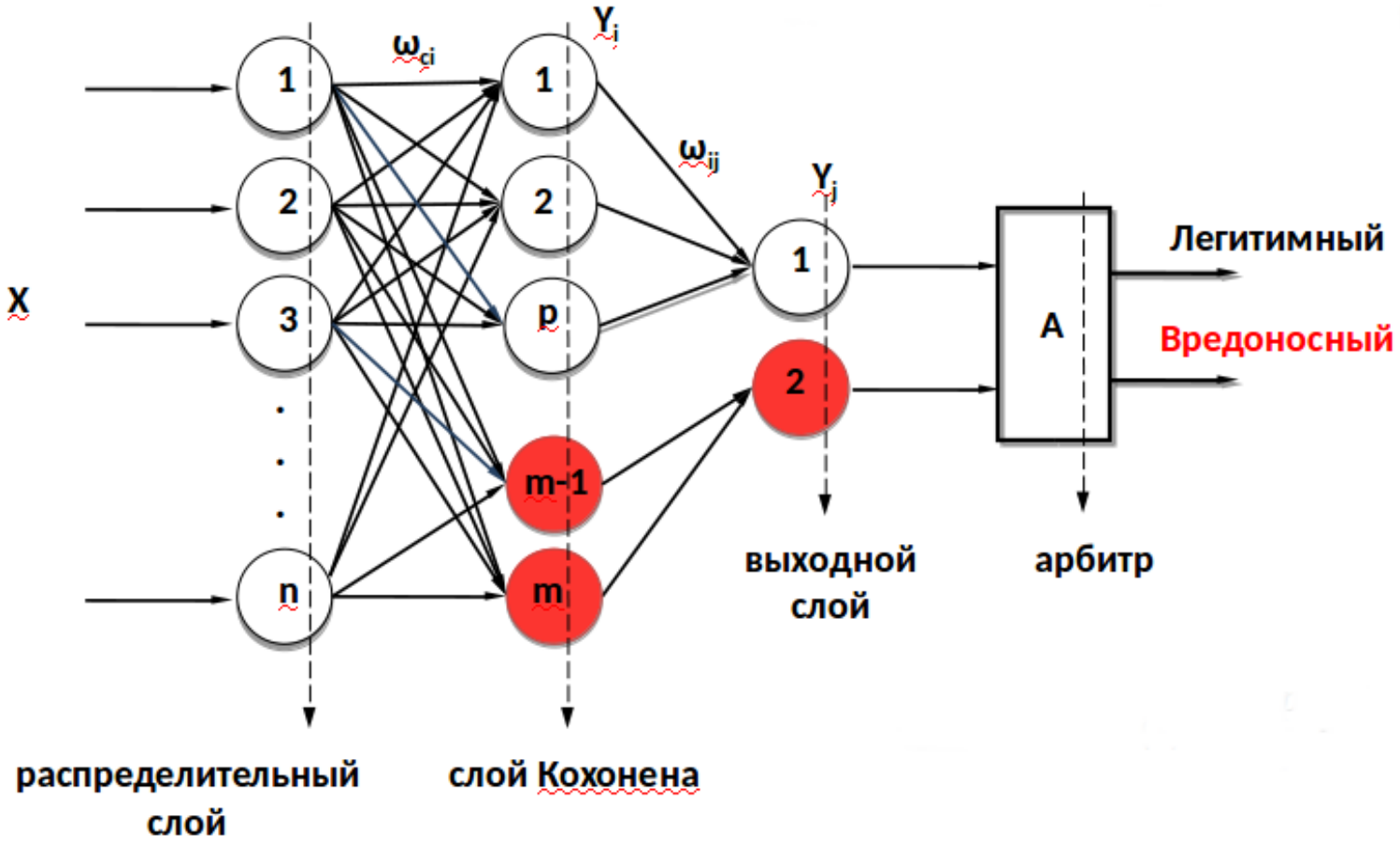
где Z_{ij}^k – значение j-го выхода i-го детектора при подаче на сход его k-го образа.
Величина суммарной квадратичной ошибки характеризует приспособленность детектора к обнаружению вредоносных файлов. Чем меньше ее значение, тем больше приспособленность детектора. Поэтому величину суммарной квадратичной ошибки можно использовать для отбора лучших детекторов. Набор обученных детекторов образует популяцию, которые циркулируют в системе и производят обнаружение вредоносных программ.

Алгоритм функционирования

- Генерация начальной популяции иммунных детекторов
- Обучение сформированных иммунных нейросетевых детекторов. Обучающая выборка формируется случайным образом.
- Отбор (селекция) нейросетевых иммунных детекторов на тестовой выборке. Если квадратичная ошибка не равна 0, такой детектор уничтожается.
- Каждый детектор наделяется временем жизни и случайным образом выбирает файл для сканирования из совокупности файлов, которые он не проверял.
- Сканирование каждым детектором выбранного файла, в результате которого определяются выходные значения детекторов $Z_{i1}, Z_{i2}, i = 1, r$.
- Если i-й детектор не обнаружил вирус в сканируемом файле, т.е. $Z_{i1} = 1$ и $Z_{i2} = 0$, то он выбирает следующий файл для сканирования. Если время жизни i-го детектора закончилось, то он уничтожается, вместо него генерируется новый детектор

- Если i-й детектор обнаружил вирус в сканируемом файле, т.е. $Z_{i1} = 0$ и $Z_{i2} = 1$, то подается сигнал об обнаружении вредоносного файла и осуществляются операции клонирования и мутации соответствующего детектора. Операция мутации заключается в дополнительном обучении детекторов-клонов на обнаруженном вредоносном файле. Так создается совокупность детекторов, настроенных на обнаруженную вредоносную программу.
- Отбор клонированных детекторов, которые являются наиболее приспособленными к обнаружению вредоносной программы. Если $E_{ij} < E_i$, то детектор прошел отбор. Здесь E_{ij} – суммарная квадратичная ошибка j-го клона i-го детектора, которая вычисляется на вредоносном файле.
- Детекторы-клоны осуществляют сканирование файлового пространства компьютерной системы до тех пор, пока не произойдет уничтожение всех проявлений вредоносной программы.
- Формирование детекторов иммунной памяти. На этой итерации определяются нейросетевые иммунные детекторы, показавшие наилучшие результаты при обнаружении присутствующего в компьютерной системе вируса. Детекторы иммунной памяти находятся в системе достаточно длительное время и обеспечивают защиту от повторного заражения.

Структура и алгоритм обучения нейросетевого детектора



Структура. Пояснение

На вход такого детектора в режиме функционирования подаются фрагменты проверяемого файла, которые формируются в соответствии с методом скользящего окна. Первый слой нейронных элементов является распределительным. Он распределяет входные сигналы на нейронные элементы второго (скрытого) слоя. Количество нейронных элементов распределительного слоя равняется размерности скользящего окна. Второй слой состоит из нейронов Кохонена, которые используют конкурентный принцип обучения и функционирования в соответствии с правилом «победитель забирает все».
Третий слой состоит из двух линейных нейронных элементов, которые используют линейную функцию активации. Арбитр осуществляет процедуру окончательного решения о принадлежности сканируемого файла к вирусному или чистому классу.
Количество нейронов слоя Кохонена равняется m. Причем $m = p + r$, где p – количество первых нейронов слоя Кохонена, которые соответствуют классу чистых программ; r – количество последних нейронов слоя Кохонена, активность которых характеризует класс вредоносных программ.

Алгоритм обучения слоя Кохонена

- Случайная инициализация весовых коэффициентов нейронов слоя Кохонена.
- Подается входной образ из обучающей выборки на нейронную сеть и производятся следующие вычисления:
  - вычисляется Евклидово расстояние между входным образом и весовыми векторами нейронных элементов слоя Кохонена
  - определяется победитель с номером k
  - производится модификация весовых коэффициентов нейрона-победителя
- Процесс повторяется, начиная с пункта 2 для всех входных образов.

Роль арбитра

Окончательное решение о том, является ли сканируемый файл вирусным, принимает арбитр.
Он вычисляет количество чистых и вредоносных фрагментов файла в соответствии с выражениями:

$$\tilde{Y_1} = \sum_{k=1}^{L} Y_1^k$$

$$\tilde{Y_2} = L - \tilde{Y_1} = \sum_{k=1}^{L} Y_2^k$$

Y_i^k – выходное значение i-го нейрона линейного слоя, L – множество образов сканируемого файла
Вычисляются вероятности принадлежности сканируемого файла соответственно к чистому и вредоносному классу

$$P_t = \frac{\tilde{Y_1}}{L} * 100\%; P_f = 1 - P_t = \frac{\tilde{Y_2}}{L} * 100\%$$

Роль арбитра

Окончательно решение принимается следующим образом:

$$Z_1 = \begin{cases} 1, \text{если} P_t > 80\% \\ 0, \text{иначе} \end{cases}$$

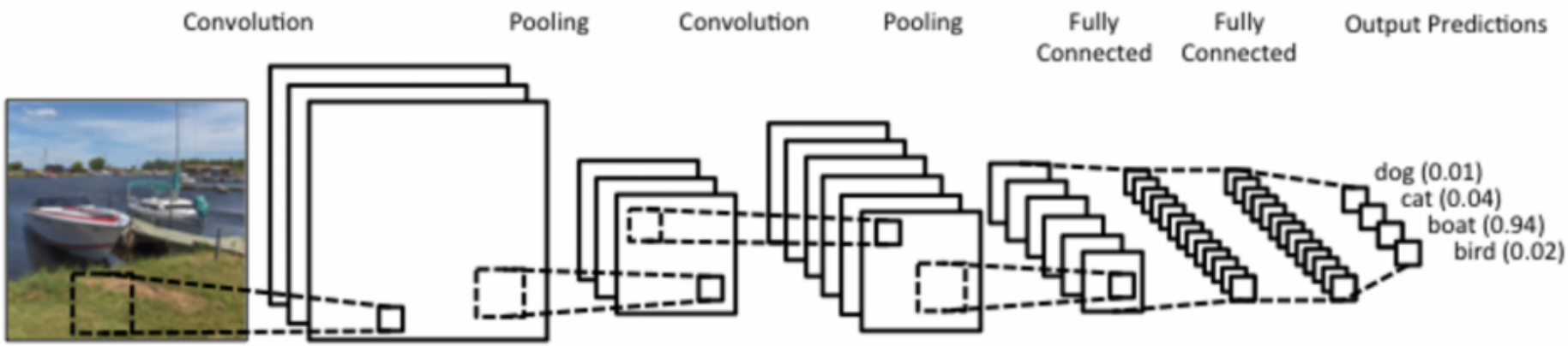
Для вредоносного файла

$$Z_2 = \begin{cases} 1, \text{если} P_f > 20\% \\ 0, \text{иначе} \end{cases}$$

Результаты испытаний

Имя файла	Антивирус Касперского (актуал. базы)	Антивирус Касперского (устар. базы)	NOD32 (эвристическ. анализатор)	ИИС (на основе 4-х детекторов)
Backdoor.Win32.Agent.lw	*Backdoor*	OK	OK	OK
Backdoor.Win32.Agobot	*Backdoor*	Backdoor	Win32/Agobot	Вирус
Email-Worm.BAT.Maddas	*Email-Worm*	Email-Worm	OK	Вирус
Email-Worm.JS.Gigger	*Email-Worm*	Email-Worm	OK	Вирус
Email-Worm.VBS.Loding	*Email-Worm*	Email-Worm	OK	Вирус
Email-Worm.Win32.Zafi.d	*Email-Worm*	OK	NewHeur_PE	Вирус
Net-Worm.Win32.Bozori.a	*Net-Worm*	OK	Win32/Bozori	Вирус
Net-Worm.Win32.Mytob.a	*Net-Worm*	OK	Win32/Mytob	Вирус
Trojan-Downl.JS.Psyme.y	*Trojan*	OK	OK	Вирус
Trojan-Downl.Win32.Bagle	*Trojan*	OK	Win32/Bagle	Вирус
Trojan-Proxy.Daemonize	*Trojan*	Trojan	OK	OK
Trojan-Proxy.Mitglieder	*Trojan*	Trojan	Win32/Trojan	Вирус
Trojan-Proxy.Win32.Agent	*Trojan*	Trojan	OK	Вирус
Trojan-PSW.LdPinch	*Trojan*	Trojan	Win32/PSW	Вирус
Virus.Win32.Gpcode.ac	*Virus.Win32*	OK	OK	Вирус

Сверточные нейронные сети



Сверточные нейронные сети

- Применяются главным образом при **анализе изображений**: в задачах классификации, сегментации, детекции и т.д. Но могут с успехом быть применены и к более разнообразным данным
- В отличие от многослойных персептронов, в которых используется линейная модель представления признаков на вход нейронной сети, при подаче данных на СНС все изображение разбивается на участки **прямоугольной формы**. Такой слой называется **сверточным (convolutional layer)**. Подобный подход позволяет учитывать локальные пространственные конфигурации признаков.
- Благодаря наличию т.н. **подвыборочных слоев (pooling layer)**, подобные сети способны формировать **инвариантные** к геометрическим преобразованиям признаки, которые потом легко распознаются традиционными классификаторами
- Существует большое количество разных сверточных нейронных сетей. Они отличаются как по архитектуре, так и по назначению. Также встречаются сети-гибриды, например сверточные рекуррентные сети

Сверточный слой

При подаче обучающего образа на сверточный слой НС выполняется разбиение входного изображения на двумерные фрагменты, совпадающие с размером фильтра на сверточном слое. Каждый фрагмент поэлементно умножается на фильтр, суммируется и помещается в соответствующую позицию выходной карты признаков. Количество входных карт и фильтров на сверточном слое варьируется. Таким образом, выход сверточного слоя l определяется следующим образом:

$$y_{ij}^{l} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} w_{ab} y_{(i+a)(j+b)}^{l-1} + b_{ab},$$

где $y_{(i+a)(j+b)}^{(l-1)}$ – выход предыдущего слоя l размерностью $N \times N$, w_{ab} – фильтр размерностью $m \times m$, b_{ab} – пороговый элемент. Таким образом, размерность выходной карты признаков составит $(N - m + 1) \times (N - m + 1)$. Если есть параметр **stride**, задающий шаг свертки, то выходная размерность определяется как $((N - m)/s + 1) \times ((N - m)/s + 1)$, где s – величина шага свертки.

Сверточный слой

Для краткости можно записать выход сверточного слоя таким образом:

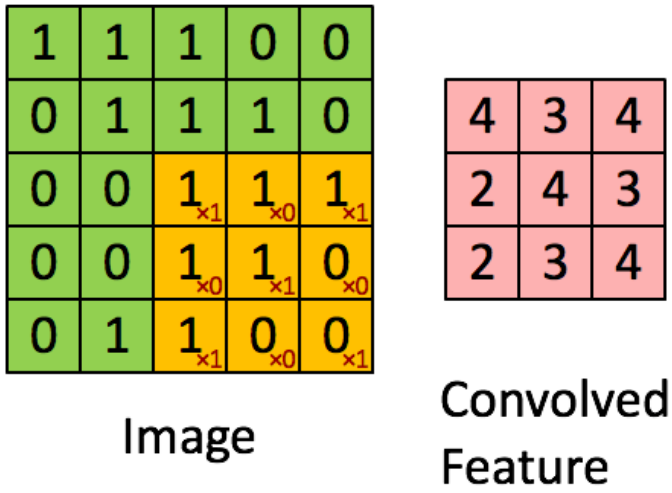
$$Y^l = Y^{l-1} * W + b,$$

где символ $*$ служит для обозначения операции свертки. Перепишем эту формулу, учитывая тот факт, что сверточный слой может включать несколько фильтров, а также то, что на слой может подаваться несколько карт одновременно:

$$Y_i^l = \sum_{j=1}^{n} Y_j^{l-1} * W_{ji}^{l-1} + b_i^{l-1},$$

где n – число подаваемых карт.
Неплохой источник: `http://cs231n.github.io/convolutional-networks/`

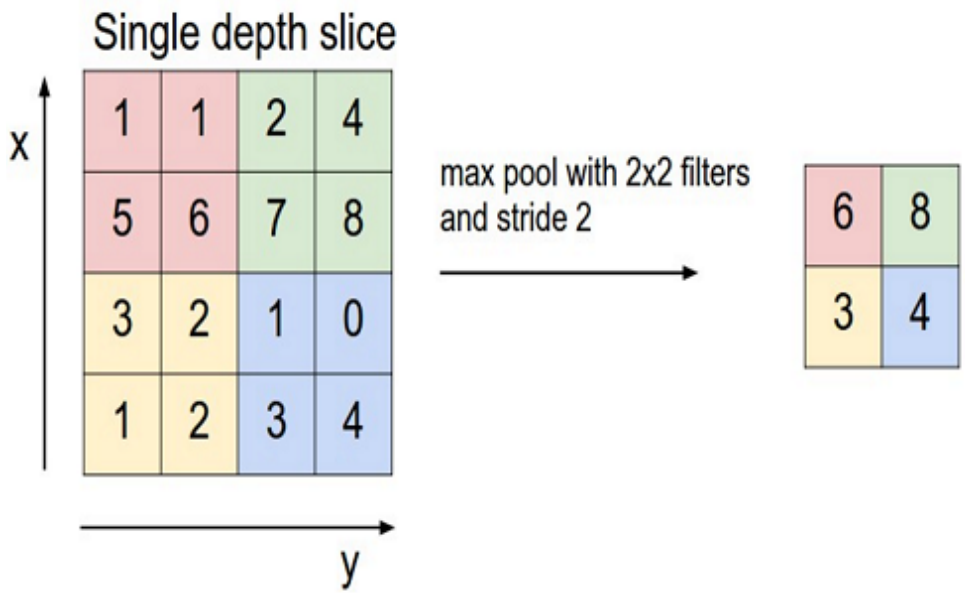
Иллюстрация свертки



Источник: `http://mourafiq.com/2016/08/10/playing-with-convolutions-in-tensorflow.html`

Подвыборочный слой

Подвыборочный слой не содержит настраиваемых параметров и имеет простую структуру. Входная карта признаков размерностью $N \times N$ разбивается на $\frac{N}{k} \times \frac{N}{k}$ одинаковых областей размерностью $k \times k$, к каждой из которых применяется редуцирующая размерность функция (например, максимум или среднее арифметическое). Среди наиболее часто используемых подвыборочных слоев выделяют **max pooling layer** и **average pooling layer**.



Источник: `https://adeshpande3.github.io/`

Слой нелинейного преобразования

После вычисления всех элементов карты к ней применяется нелинейное преобразование. Данный слой представляет собой абстракцию для нелинейного преобразования входных карт. Фактически здесь применяется функция активации к **каждому элементу каждой карты**, после чего преобразованные данные передаются дальше по сети.

$$y_{ij}^{l} = \sigma\left(\sum_{a=0}^{m-1}\sum_{b=0}^{m-1} w_{ab} y_{(i+a)(j+b)}^{l-1} + b_{ab}\right),$$

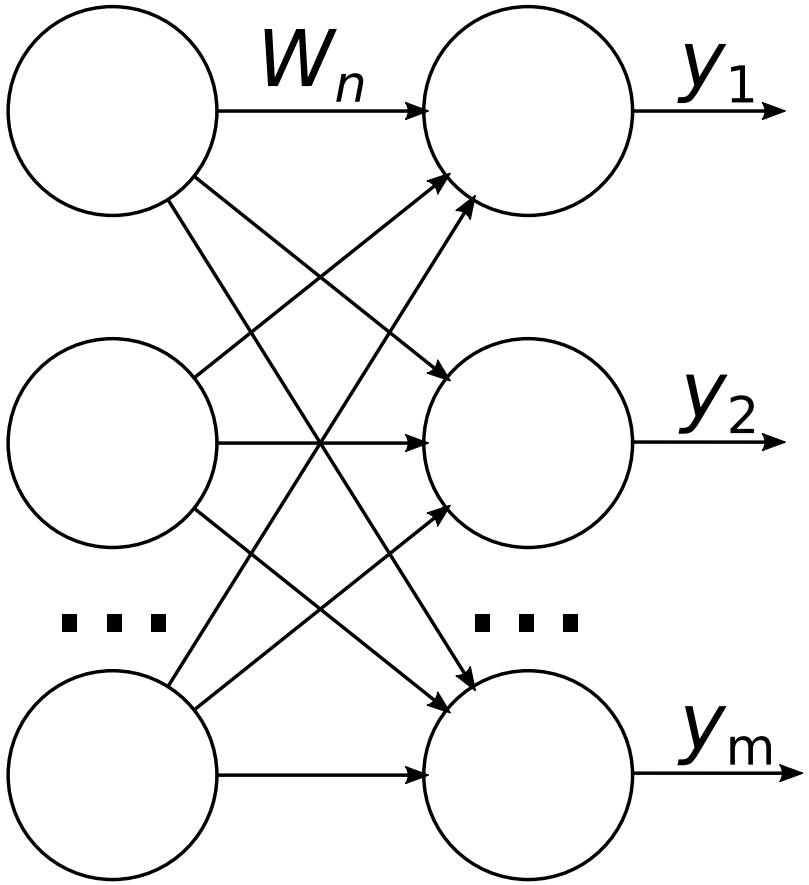
Слой нелинейного преобразования не изменяет размерность!

Особое значение имеет ReLU-функция активации, которая помогла решить проблему исчезающего градиента, когда в процессе обучения первые слои нейронной сети «затормаживали» свое обучение.

$$f(x) = \max(0, x)$$

Полносвязный слой

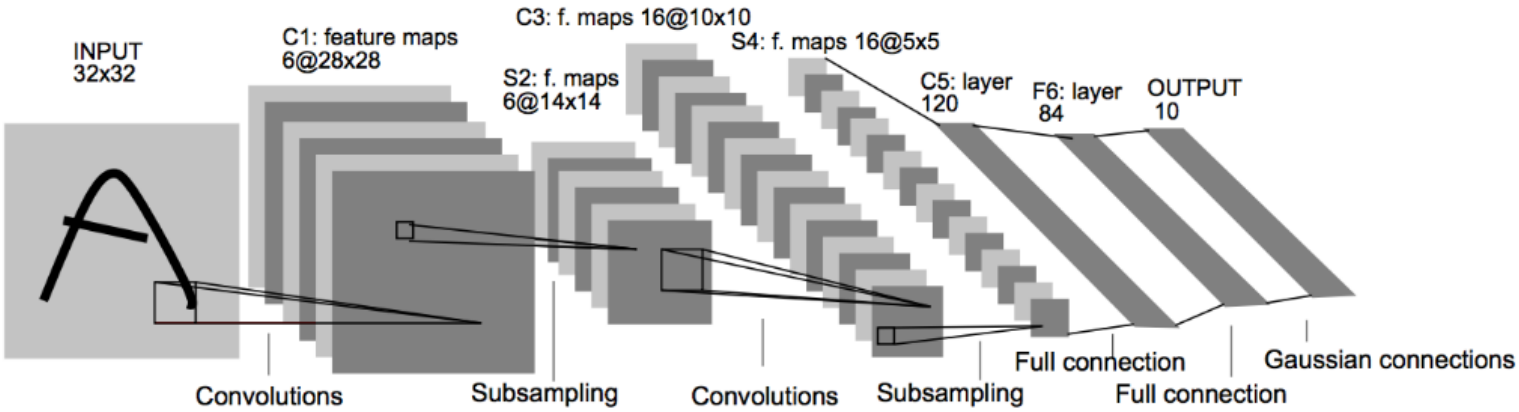
Этот слой полностью соответствует слоям, являющихся основным «строительным» блоком для многослойных персептронов.



DropOut-слой

Данный тип слоев используется для препятствия сети переобучению. Часть значений после активации отбрасывается, тем самым сеть хуже приспосабливается к входным данным, но при этом улучшается ее обобщающая способность. Сеть становится более избыточной.
DropOut используется только на этапе обучения!

Классическая архитектура CHC: LeNet-5



CNN called LeNet by Yann LeCun (1998)

Источник: `http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/`

Задачи, решаемые сверточными нейронными сетями

- Распознавание изображений
- Детекция (локализация) объектов на изображениях
- Сегментация изображений

Домашнее задание

- Изучить выборку Forest Fires (`https://archive.ics.uci.edu/ml/datasets/Forest+Fires`). Решить регрессионную задачу на прогнозирование площади леса, поврежденного лесными пожарами (с учетом замечаний по улучшению модели, сделанных на прошлом занятии).
- Построить прогнозирующую нейронную сеть на базе обычного многослойного персептрона и сравнить ее характеристики с рекуррентными нейронными сетями (можно использовать предложенную реализацию или сделать свою).
- Организовать простейшую сверточную сеть (типа LeNet-5) с использованием фреймворка Caffe и обучить ее классификации образов из выборки MNIST. Сравнить результаты с полученными ранее для многослойного персептрона с предобучением и без.