УНИВЕРСИТЕТ
искусственного
интеллекта

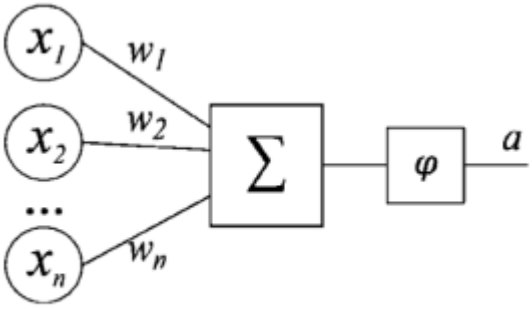
Оценка качества обучения нейронной сети

занятие 2

Оценка качества обучения нейронной сети

Занятие № 2

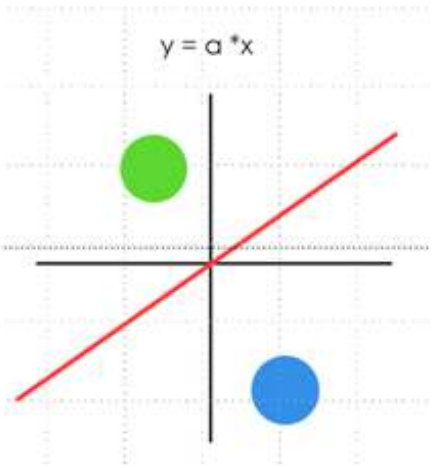
BIAS в нейронах



Bias – порог или нейрон смещения. Что это такое и для чего нужно?

Модель нейрона, которую вы должны помнить из прошлого занятия:

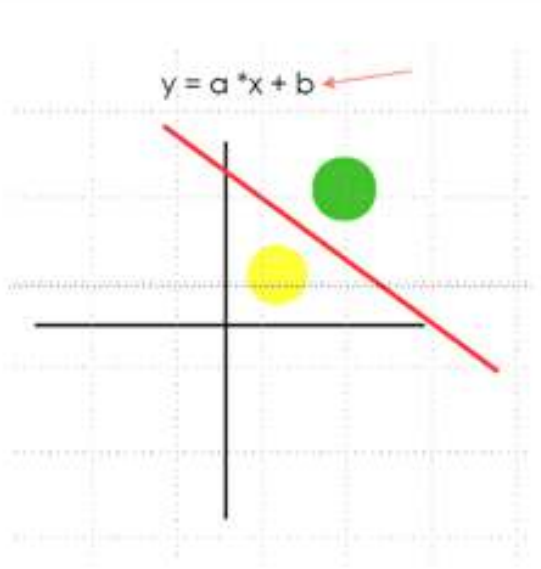
Полная формула выглядит так N=x1*w1+x2*w2+xn*wn+W. Последняя +W и есть Bias. Зачем он нужен?



В упрощенном варианте, задача нейронной сети – провести разделяющую плоскость (в данном случае, прямую) между 2-мя классами.

При таком расположении классов прямая проходит через ноль и проблем не возникнет.

BIAS
в нейронах



А в этом случае без смещения не обойтись. Разделить два класса, проведя разделяющую плоскость через ноль, невозможно. Для этого нейрон смещения и нужен.

Keras автоматически добавляет нейрон смещения в слои. При желании его добавление можно отключить. Достаточно указать в параметрах слоя use_bias=False.

Обучающая, проверочная и тестовая выборки

Обучающая выборка (training set) – набор данных, который используется для обучения сети. Точность на ней может достигать 100%.

Проверочная (валидационная) выборка (validation set) – набор данных, который используется в процессе обучения для оценки качества обучения. Как правило, точность ниже, чем на обучающей выборке. Ее часто путают с тестовой выборкой. Обычно создается выделением части обучающей выборки для проверки работы модели.

Тестовая выборка (test set) – набор данных, который используется для оценки качества работы сети после завершения обучения. Используется для контрольной финальной проверки перед запуском в продакшн.

В уроках обычно используются обучающая и проверочные выборки, без тестовой. При упоминаниях точности на тестовой, обычно имеется в виду проверочная.

В целом у нейронной сети есть 2 типа параметров. Изменяемые в процессе обучения и неизменяемые «Гиперпараметры модели».

Параметры модели (на эти параметры может влиять сама модель):

- изменяются в процессе обучения;
- для нейронной сети – веса входов в нейроны.

Обучающая, проверочная и тестовая выборки

Гиперпараметры модели:

- не меняются сетью в процессе обучения;
- влияют на конфигурацию модели и методы обучения;
- для нейронной сети – количество слоев, количество нейронов на слое, регуляризация, скорость обучения, размер мини-выборки.

Гиперпараметры изменяются перед запуском обучения. Для изменения количества слоев и нейронов в слое нужно пересоздавать модель, т.е. веса модели обнулятся. Скорость обучения и размер выборки можно менять перед очередным этапом обучения без сброса весов. А скорость обучения можно менять даже в процессе обучения с помощью callback-ов (о них вам расскажут в одной из следующих лекций).

Процесс создания нейронной сети при наличии подготовленных данных сводится к простому алгоритму. Установка гиперпараметров -> обучение сети на обучающей выборке -> проверка работы на валидационной выборке. И так по кругу, до достижения необходимых показателей. После этого нужна проверка на тестовой выборке для подтверждения результатов и сеть готова. Конечно, работы перед запуском в production будет еще много, но это тема другого курса. С темами курса «Интеграция AI решений в production» вы можете ознакомиться по ссылке https://neural-university.ru/kurs_integration

Наборы данных для обучения

В уроке будут использоваться 2 базы: sonar.csv – данные с эхолота и cars_new.csv – база машин с Юлы.

Подгружаем необходимую библиотеку «from google.colab import files» и с помощью files.upload() загружаем базы.

Можно загружать данные с google drive, для этого понадобится импортировать библиотеку работы с гугл диском и подключить сам диск

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

Наборы данных для обучения

После этого можно обращаться к файлам на диске.

С помощью pandas читаем загруженные данные

```
df = pd.read_csv("sonar.csv", header=None)
```

Если используем гугл драйв, то нужно указать путь к файлу

```
df = pd.read_csv("drive/My Drive/sonar.csv", header=None)
```

База с эхолота представляет собой таблицу из 208 строк и 61 столбца. 60 столбцов представляют собой данные с датчиков. А 61-ый столбец – это результат в виде букв R(rock – камень) и M (Мина). Эти буквы нам нужно будет преобразовать в числовые данные, т.к. нейронная сеть работает с числами.

Посмотрев на данные, можно понять, что на вход нейронной сети будет подаваться вектор из 60 элементов, а на выходе получим результат в виде 0 (если это камень) или 1 (если мина).

Преобразуем данные. В “X” пойдут значения из 60 столбцов (тип float), в “У” данные 61-го столбца, преобразованные в 0 и 1 (тип int).

После преобразования имеем 2 вектора размерностями (208,60) и (208,)

Посмотрев вектор «У», видим, что у нас есть 104 записи для камней и за ними 104 записи для мин.

Создание тренировочной и проверочной выборки

Приступаем к созданию тренировочной и проверочной выборки. Для этого нам понадобится библиотека **sklearn** и функция **train_test_splite** (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

Подключаем библиотеку:

```
from sklearn.model_selection import train_test_split
```

И делаем разбивку на тренировочную и тестовую выборки:

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, shuffle=True)
```

Создание тренировочной и проверочной выборки

Функция train_test_split – в нее нам необходимо передать «X» и «У», указать параметр test_size – В каком соотношении нам нужно разбить выборку. И в данном случае нам нужно использовать shuffle=True, т.к. необходимо перемешать данные. Соответствие элементов X и У останется. Например, x[1] станет x[93], так же как и y[1] станет y[93]. Функция вернет нам 4 переменные, т.е. 2 набора, x_train, x_test, y_train и y_test.

Вместо одной переменной «X» размером (208,60) мы получили 2 переменных размерностью (166, 60) и,(42, 60). Так же с «У», была одна на 208, стало 2 по 166 и 42.

Создадим простую сеть:

```
model = Sequential()

# Добавляем слои
model.add(Dense(60, input_dim=60, activation='relu'))
model.add(Dense(30,  activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

В первом слое указываем размерность входных данных input_dim=60. При этом количество нейронов не обязательно должно быть 60, т.е. первый слой мог бы выглядеть и так:

```
model.add(Dense(20, input_dim=60, activation='relu'))
```

На выходном слое 1 нейрон. В качестве функции активации используем sigmoid – эта функция активации лучше всего подходит для вычисления вероятностей, т.к. принимает значение от 0 до 1. И, получив на выходе 0.7214, мы будем знать, что с вероятностью 72% входные данные соответствуют мине.

Компилируем модель:

```
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.001), metrics=['accuracy'])
```

Используем **binary_crossentropy** для функции ошибки, т.к. в данном случае это задача двоичной классификации (https://keras.io/api/losses/).

Указываем метрику **metrics=['accuracy']**

Создание тренировочной и проверочной выборки

Указываем оптимайзер, в данном случае **Adam**, и устанавливаем скорость обучения (learning_rate) **lr=0.001**. Такую скорость можно было не указывать, т.к. это скорость обучения по умолчанию. Скорость обучения — это шаг, на который изменяются веса. Поэтому при значении lr=0.1 обучение будет идти быстрее, но снизится точность.

Предположим, для 100% точности значение какого-то из весов должно стать 0.2391, а при создании сети этот вес был 0.5. При шаге 0.1 мы очень быстро приблизимся к значению 0.2, но сделать даже 0.23 мы уже не сможем, т.к. шаг слишком большой. А при lr=0.00001 потребуется очень много шагов и, соответственно, времени на обучение. Поэтому высокую скорость обучения выгодно использовать на первых этапах обучения, а потом постепенно снижать для увеличения точности. Для маленьких нейронок это не актуально, а для более тяжелых, когда время на эпоху составляет более 5 минут (а нужно 500 эпох), может существенно экономить время. Это можно делать вручную либо использовать методы динамического изменения скорости обучения (вы их будете изучать позже).

Для ручного изменения скорости обучения нужно перекомпилировать (не пересоздать) модель. В этом случае веса сохранятся и можно спокойно обучать модель дальше. Выглядит это примерно так:

```
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.1), metrics=['accuracy'])
model.fit(...)
model.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.001), metrics=['accuracy'])
model.fit(...)
```

Запускаем обучение:

```
model.fit(x_train, y_train, batch_size=8, epochs=100, verbose=1)
```

База для обучения маленькая, поэтому **batch_size** ставим небольшим. За 100 эпох обучения сеть достигает 100% точности на обучающей выборке. Проблема в том, что все эти данные сеть видела и веса подгонялись именно под их. Поэтому далеко не факт, что, получив другие данные, она сработает с такой же точностью. Как проверить?

Для этого нам необходимо создать проверочную выборку. Есть несколько способов это сделать.

Создание тренировочной и проверочной выборки

Способ первый. Добавить в параметры функции обучения **validation_split:**

```
model.fit(x_train, y_train, batch_size=8, epochs=100, validation_split=0.2,
          verbose=1)
```

В этом случае функция обучения сама выделит часть выборки под проверочную. Казалось бы, хороший и простой способ, но есть недостатки. Если вы повторно запустите функцию обучения, например, не хватило эпох для нужного результата, разбивка на обучающую и проверочную выборки будет проведена заново. Чем это грозит? В обучающую выборку попадут значения из проверочной и наоборот. Весь смысл разделения выборок потеряется.

Способ второй. Вручную выделить часть выборки с помощью среза:

```
n_val = 30
model.fit(x_train[:-n_val], y_train[:-n_val], batch_size=8, epochs=100,
          validation_data=(x_train[-n_val:], y_train[-n_val:]),
          verbose=1)
```

В данном случае мы выделяем в проверочную выборку 30 значений с конца массива. И нам нужно указать **validation_data**, т.к. проверочные данные мы задаем явно. В некоторых случаях подходит только этот способ, особенно при выделении проверочной выборки для временных рядов.

Третий способ. Опять использовать **train_test_splite**. Но вместо X и У разделить x_train и y_train:

```
x_train_new, x_val, y_train_new, y_val = train_test_split(x_train, y_train,
                                                          test_size=0.2)
model.fit(x_train_new, y_train_new, batch_size=8, epochs=100,
          validation_data=(x_val, y_val), verbose=1)
```

Какой бы способ мы не использовали, запустив обучения, мы увидим, что вывод информации об обучении изменился. Появились данные по проверочной выборке. И точность на проверочной выборке ниже, чем на обучающей. Но эта точность уже ближе к реальной.

Создание тренировочной и проверочной выборки

На лекции точность на проверочной получалась порядка 85%. Теперь проверим точность на тестовом наборе. Для этого нам понадобится метод **evaluate:**

```
scores = model.evaluate(x_test, y_test, verbose=1)
```

Подаем на вход наш тестовый набор и получаем результат. На лекции точность получилась еще ниже, чем на проверочной выборке, 73%.

Это и есть реальная точность нашей сети. Если она нас устраивает, все отлично. Если нет, возвращаемся к началу: меняем архитектуру, экспериментируем со слоями, количеством нейронов, активационными функциями, возможно, ищем, как расширить базу и т.д.

Визуализация процесса обучения

Мы можем построить графики процесса обучения. В первую очередь, для этого нам нужно сохранить результаты. Делается это так:

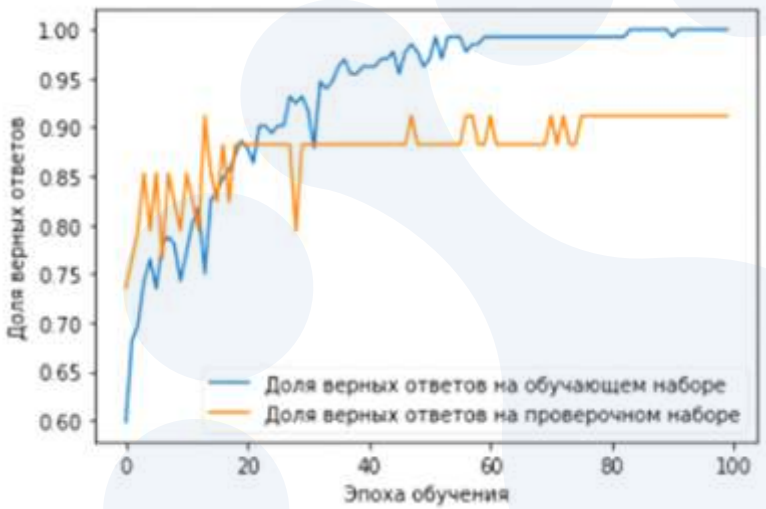
```
history = model.fit(...)
```

В history запишется словарь с ключами ['loss', 'accuracy', 'val_loss', 'val_accuracy'], где

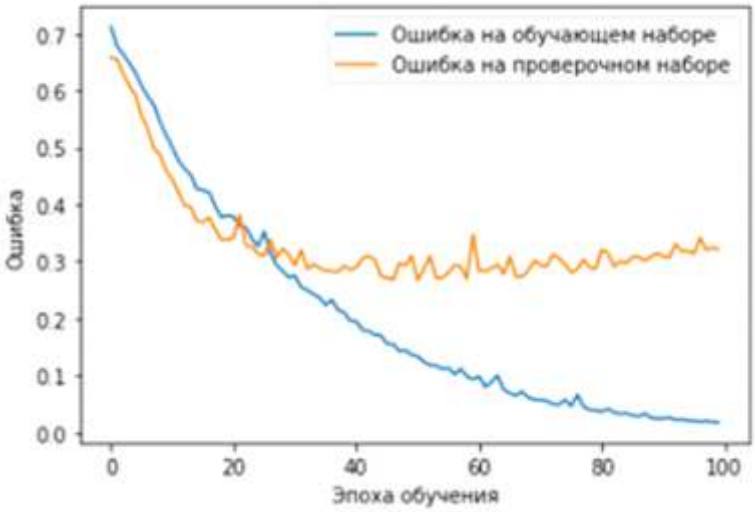
- 'loss' – значения ошибки на обучающем наборе данных;
- 'accuracy' – точность на обучающем наборе данных;
- 'val_loss' – значения ошибки на проверочном наборе данных;
- 'val_accuracy' – точность на проверочном наборе данных.

Теперь мы можем построить графики обучения для каждого параметра. Для построения графика можно использовать matplotlib:

Это вполне нормальный график обучения, причин для беспокойства нет.



Визуализация процесса обучения



Но если посмотреть на график loss, можно заметить, что примерно с 50-ой эпохи график ошибки на обучающей выборке продолжает падать, а график ошибки на проверочной выборке начинает незначительно, но стабильно повышаться, а это уже непорядок. Это признак переобучения.

Переобучение

Что такое переобучение? Объясняя простым языком, следует сказать, что это ситуация, когда нейронка вместо обобщения начинает заучивать правильные ответы. Выше показан именно такой пример. Примерно до 50 эпохи шло обобщение, а после пошло заучивание правильных ответов.

Почему это происходит? Есть несколько возможных причин:

- маленькая база для обучения;
- плохо собранная база;
- слишком сложная архитектура сети;
- разбалансировка базы.

В нашем примере причинами могли быть маленькая база и, при некотором «везении», ее разбалансировка. База явно маленькая: сравните базу эхолота и mnist, 208 записей против 60000.

Разбалансировка: изначально база эхолота небольшая, но хорошо сбалансированная, по 104 примера на класс. Но при разделении на выборки могла сложиться ситуация, при которой данные бы разделились неравномерно. Например, если бы мы не использовали shuffle или при случайном распределении данные бы разделились неудачно, то в обучающую выборку бы попали 104 примера с камнями и всего 22 примера с минами, а в проверочную и тестовую попали бы только мины.

Переобучение

Плохо собранная база: ошибки в базе, недостаточное разнообразие данных и т.д. Предположим, есть задача отличить русских от китайцев. Все фотографии в России делались в деревнях, а китайцев – в крупных городах. Скорее всего, в этом случае нейронка вместо лиц будет смотреть на фон. И всех людей в городах посчитает китайцами, а в деревнях – русскими.

Слишком сложная архитектура: дело, скорее, не просто в архитектуре, а в архитекторе относительно количества данных. Если нейронке хватит мощности просто запомнить весь набор, она именно это и сделает. Поэтому мощность нейронки стоит наращивать постепенно.

Как бороться с переобучением?

- Увеличивать базу,
- чистить базу,
- упрощать архитектуру,
- упрощать параметризацию,
- нормализовать данные,
- использовать Dropout,
- использовать BatchNormalization.

Dropout

Слой Dropout отключает часть нейронов слоя, чтобы нейроны не заучивали признаки. Параметр слоя отвечает за то, какая часть нейронов будет отключена.

```
model.add(Dense(100, activation='relu'))
model.add(Dropout(0.4))
```

В этом примере на каждой эпохе обучения активными будут только 60 нейронов из 100, а указанные 0.4(40%) будут выключены. На каждой эпохе это будет разный набор нейронов. Особенность слоя Dropout в том, что он действует только при обучении, при обычной работе сети он отключается и задействуются все нейроны слоя.

http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

BatchNormalization

BatchNormalization

Нейронные сети обычно лучше работают с данными от 0 до 1. Слой **BatchNormalization** нужен для приведения данных к нормальному распределению. Использовать его очень просто:

```
model.add(Dense(60, activation='relu'))
model.add(BatchNormalization())
```

На самом деле, у этого слоя огромное количество параметров, но чаще всего можно оставить все по умолчанию. Не всегда этот слой нужен, а иногда даже ухудшает точность. Но может и существенно помочь.

Этот слой иногда может помочь в случаях градиентного взрыва.

Пример из практики: сеть из нескольких блоков одномерных сверток с активацией Relu.

```
model.add(Conv1D(180, 5, activation='relu', padding='same'))
model.add(Conv1D(180, 10, activation='relu', padding='same'))
model.add(Flatten())
model.add(BatchNormalization())
model.add(Dense(7, activation='sigmoid',
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l2(0.01)))
```

Перед слоем с активацией sigmoid значения превышали допустимый лимит и происходил градиентный взрыв. Без слоя BatchNormalization сеть через несколько десятков эпох обучения вставала на ошибку, loss и accuracy уходили в nan. Добавление BatchNormalization решило проблему, и сеть стала обучаться как положено.

Подготовка данных для работы с базой машин с Юлы

Загрузив базу, видим, что в базе много текстовых полей, цифры далеки от желаемых 0..1, и не совсем понятно, как с этим работать.

Нам потребуется преобразовать данные, чтобы нейронка могла с ними работать.

Подготовка данных для работы с базой машин с Юлы

Марка, модель, тип кузова, кпп, тип топлива – это списки. Если мы просто поставим номер по списку, то может получиться ситуация, что вес входа бензина будет больше дизеля, универсал важнее джипа, а мерседес не так важен, как Kia. Такие данные лучше всего переводить в Ohe (One Hot Encoding). Что это за формат? Это массив из 0 и 1 Например, такой [0,0,0,1]. Позиция 1 в таком массиве соответствует индексу списка. [0,0,0,1] будет соответствовать индексу 3 (нумерация идет с нуля). Для нейросети такое представление данных предпочтительнее, т.к. все индексы будут иметь изначально одинаковый вес.

Для перевода в Ohe создаются словари, где значения соответствуют индексу, и далее индекс переводится в Ohe.

Остальные данные нам нужно привести к стандартному распределению с помощью preprocessing.scale. Почти то же самое делаем с ценой.

С этими данными нейросеть уже сможет работать.