1 Лабораторная работа 1

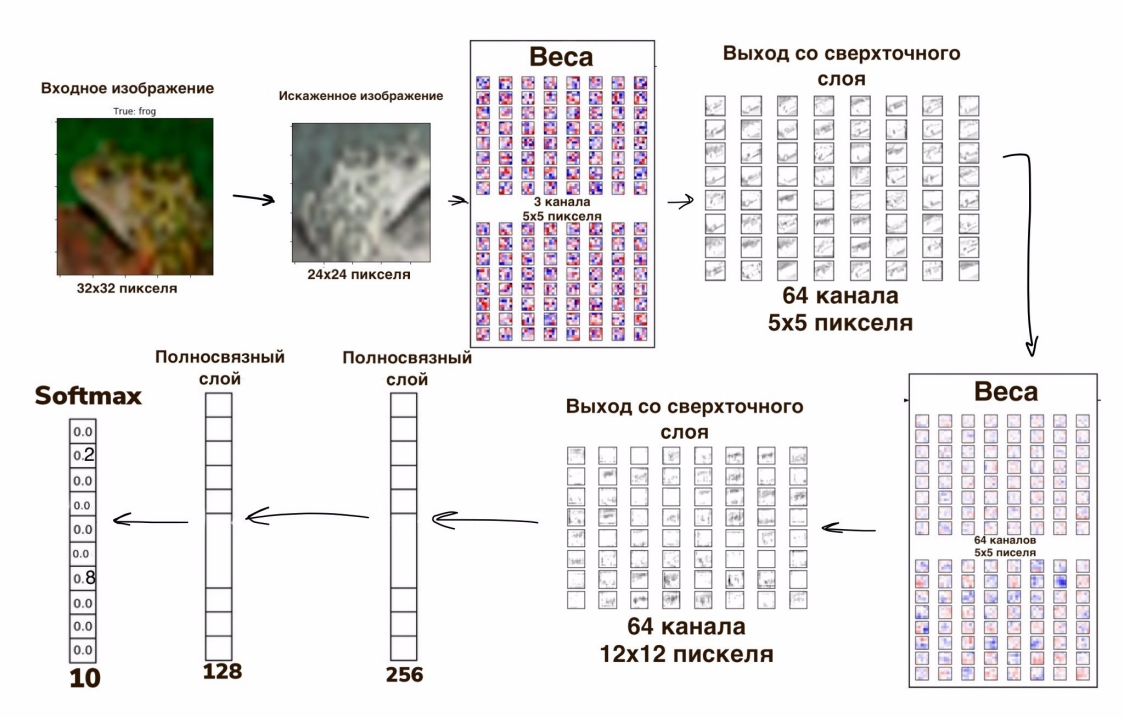
1.1 Классификация изображений

1.2 Введение

В этом примере показано, как создать простой классификатор изображений на основе сверточной нейронной сети и наборе данных CIFAR-10.

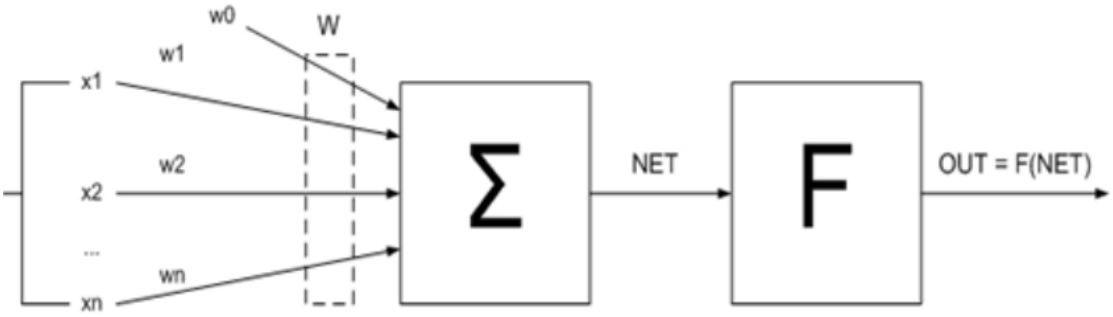
1.3 Структура сверточной нейронной сети

СНС состоит из разных видов слоев: сверточные (convolutional) слои, субдискретизирующие (subsampling, pooling) слои и полносвязный слои.



Первые два типа слоев (convolutional, subsampling), чередуясь между собой, формируют входной вектор признаков для многослойного персептрона.

1.4 Модель нейрона



$$NET = \sum_{i=1}^{n} w_i + x_i + w_0,$$

где * w_i - вес i нейрона; * x_i - выход i нейрона;
* w_0 - смещение; * n - количество синаптических связей, входящих в нейрон.

1.5 Входной слой

В нашем случае, входные данные представляют из себя цветные изображения, размера 32x32 пикселей. Если размер будет слишком велик, то вычислительная сложность повысится, соответственно ограничения на скорость ответа будут нарушены, определение размера в данной задаче решается методом подбора. Если выбрать размер слишком маленький, то сеть не сможет выявить ключевые признаки. Каждое изображение разбивается на 3 канала: красный, синий, зеленый. Таким образом получается 3 изображения размера 32x32 пикселей.

Входной слой учитывает двумерную топологию изображений и состоит из нескольких карт (матриц), карта может быть одна, в том случае, если изображение представлено в оттенках серого.

Входные данные каждого конкретного значения пикселя нормализуются в диапазон от 0 до 1, с помощью функции нормализации:

$$f(p, min, max) = \frac{p - min}{max - min},$$

где * p - значение конкретного цвета пикселя (от 0 до 255); * min - минимальное значение пискеля;
* max - максимальное значение пикселя.

1.6 Сверточный слой

Сверточный слой представляет из себя набор карт признаков, у каждой карты есть синаптическое ядро (фильтр).

Количество карт определяется требованиями к задаче, если взять большое количество карт, то повысится качество распознавания, но увеличится вычислительная сложность. Исходя из анализа научных статей, в большинстве случаев предлагается брать соотношение один к двум,

то есть каждая карта предыдущего слоя (например, у первого сверточного слоя, предыдущим является входной) связана с двумя картами сверточного слоя.

Размер у всех карт сверточного слоя - одинаковы и вычисляются по формуле:

$$(w, h) = (mW - kW + 1, mH - kH + 1),$$

где * (w, h) - вычисляемый размер сверточной карты; * mW - ширина предыдущей карты;
* mH - высота предыдущей карты;
* kW - ширина ядра;
* kH - высота ядра.

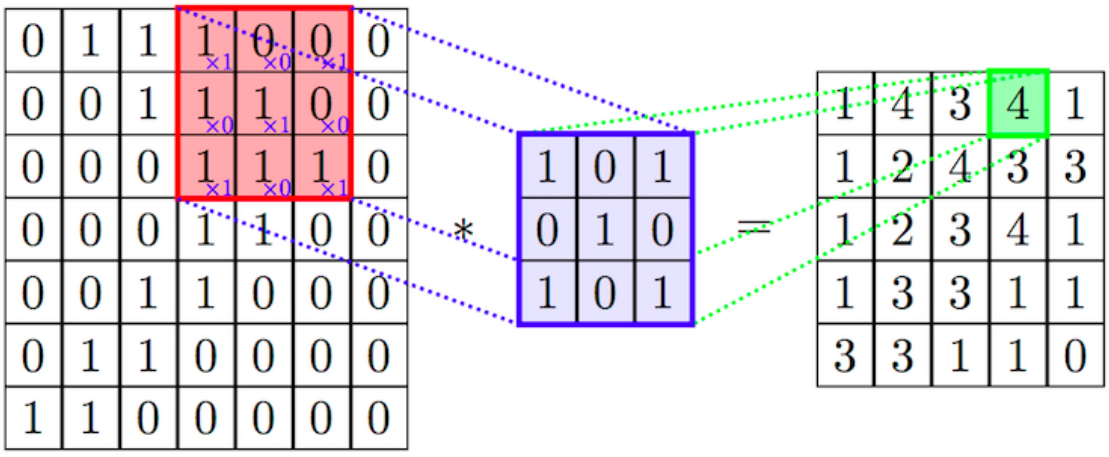
Ядро представляет из себя фильтр или окно, которое скользит по всей области предыдущей карты и находит определенные признаки объектов. Размер ядра обычно берут в пределах от 3x3 до 7x7. Если размер ядра маленький, то оно не сможет выделить какие-либо признаки, если слишком большое, то увеличивается количество связей между нейронами. Также размер ядра выбирается таким, чтобы размер карт сверточного слоя был четным, это позволяет не терять информацию при уменьшении размерности в подвыборочном слое, описанном ниже.

Ядро представляет собой систему разделяемых весов или синапсов, это одна из главных особенностей сверточной нейросети. В обычной многослойной сети очень много связей между нейронами, то есть синапсов, что весьма замедляет процесс детектирования. В сверточной сети – наоборот, общие веса позволяет сократить число связей и позволить находить один и тот же признак по всей области изображения.

Изначально значения каждой карты сверточного слоя равны 0. Значения весов ядер задаются случайным образом в области от -0.5 до 0.5. Ядро скользит по предыдущей карте и производит операцию свертка, которая часто используется для обработки изображений, формула:

$$(f * g)[m, n] = \sum f[m - k, n - l] * g[k, l],$$

где * f - исходная матрица изображения; * g - ядро свертки.

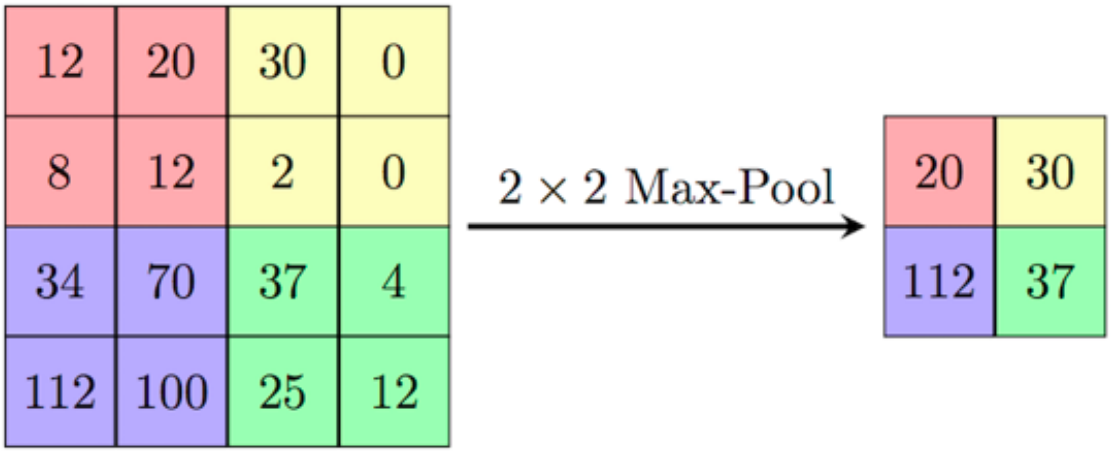


Неформально эту операцию можно описать следующим образом — окном размера ядра g проходим с заданным шагом все изображение f, на каждом шаге поэлементно умножаем содержимое окна на ядро g, результат суммируется и записывается в матрицу результата.

1.7 Подвыборочный слой

Подвыборочный слой также, как и сверточный имеет карты, но их количество совпадает с предыдущим слоем. Цель слоя – уменьшение размерности карт предыдущего слоя. Если на предыдущей операции свертки уже были выявлены некоторые признаки, то для дальнейшей обработки настолько подробное изображение уже не нужно, и оно уплотняется до менее подробного. К тому же фильтрация уже ненужных деталей помогает не переобучаться. В процессе сканирования ядром подвыборочного слоя (фильтром) карты предыдущего слоя, сканирующее ядро не пересекается в отличие от сверточного слоя. Обычно, каждая карта имеет ядро размером 2x2, что позволяет уменьшить предыдущие карты сверточного слоя в 2 раза. Вся карта признаков разделяется на ячейки 2x2 элемента, из которых выбираются максимальные по значению.

Обычно в подвыборочном слое применяется функция активации RelU. Операция подвыборки изображена на картинке:



Формально слой может быть описан формулой:

$$x^l = f(a^l * subsample(x^{l-1}) + b^l),$$

где * x^l - выход слоя l; * $f()$ - функция активации; * a^l, b^l - коэффициенты сдвига слоя l; * $subsample()$ - операция выборки локальных максимальных значений.

1.8 Полносвязный слой

Последний из типов слоев это слой обычного многослойного персептрона. Цель слоя – классификация, моделирует сложную нелинейную функцию, оптимизируя которую, улучшается качество распознавания.

Нейроны каждой карты предыдущего подвыборочного слоя связаны с одним нейроном скрытого слоя. Таким образом число нейронов скрытого слоя равно числу карт подвыборочного слоя, но связи могут быть не обязательно такими, например, только часть нейронов какой-либо из карт подвыборочного слоя быть связана с первым нейроном скрытого слоя, а оставшаяся часть со вторым, либо все нейроны первой карты связаны с нейронами 1 и 2 скрытого слоя. Вычисление значений нейрона можно описать формулой:

$$x_j^l = f(\sum_i x_j^{l-1} * w_i^{l-1} + b_j^{l-1}),$$

где * x_j^l - карта признаков j (выход слоя l); * $f()$ - функция активации; * b^l - коэффициенты сдвига слоя l; * w_i^{l-1} - матрица весовых коэффициентов слоя l.

1.9 Выходной слой

Выходной слой связан со всеми нейронами предыдущего слоя. Количество нейронов соответствует количеству распознаваемых классов, то есть в нашем случае 10.

Для уменьшения количества связей и вычислений для бинарного случая можно использовать один нейрон и при использовании в качестве функции активации гиперболический тангенс, выход нейрона со значением -1 означает, что на картинке нет распозноваемого класса, напротив выход нейрона со значением 1 – означает принадлежность к классу.

1.10 Выбор функции активации

Одним из этапов разработки нейронной сети является выбор функции активации нейронов. Вид функции активации во многом определяет функциональные возможности нейронной сети и метод обучения этой сети. Классический алгоритм обратного распространения ошибки хорошо работает на двухслойных и трехслойных нейронных сетях, но при дальнейшем увеличении глубины начинает испытывать проблемы. Одна из причин — так называемое затухание градиентов. По мере распространения ошибки от выходного слоя к входному на каждом слое происходит домножение текущего результата на производную функции активации. Производная у традиционной сигмоидной функции активации меньше единицы на всей области определения, поэтому после нескольких слоев ошибка станет близкой к нулю. Если же, наоборот, функция активации имеет неограниченную производную (как, например, гиперболический тангенс), то может произойти взрывное увеличение ошибки по мере распространения, что приведет к неустойчивости процедуры обучения.

В данной работе в качестве функции активации в скрытых и выходном слоях применяется ReLU.

Преимущества использования ReLU:

- ее производная равна либо единице, либо нулю, и поэтому не может произойти разрастания или затухания градиентов, т.к. умножив единицу на дельту ошибки мы получим дельту ошибки, если же мы бы использовали другую функцию, например, гиперболический тангенс, то дельта ошибки могла, либо уменьшиться, либо возрасти, либо остаться такой же, то есть, производная гиперболического тангенса возвращает число с разным знаком и величиной, что можно сильно повлиять на затухание или разрастание градиента. Более того, использование данной функции приводит к прореживанию весов;
- вычисление сигмоиды и гиперболического тангенса требует выполнения ресурсоемких операций, таких как возведение в степень, в то время как ReLU может быть реализован с помощью простого порогового преобразования матрицы активаций в нуле;
- отсекает ненужные детали в канале при отрицательном выходе.

Из недостатков можно отметить, что ReLU не всегда достаточно надежна и в процессе обучения может выходить из строя. Например, большой градиент, проходящий через ReLU, может привести к такому обновлению весов, что данный нейрон никогда больше не активируется. Если это произойдет, то, начиная с данного момента, градиент, проходящий через этот нейрон, всегда будет равен нулю. Соответственно, данный нейрон будет необратимо выведен из строя. Например, при слишком большой скорости обучения, может оказаться, что до 40%

ReLU никогда не активируются. Эта проблема решается посредством выбора надлежащей скорости обучения.

1.11 Импорты

```
In [1]: %matplotlib inline
        from sklearn.metrics import confusion_matrix
        import matplotlib.pyplot as plt
        from datetime import timedelta
        import prettytensor as pt
        import tensorflow as tf
        import urllib.request
        import numpy as np
        import tarfile
        import zipfile
        import pickle
        import math
        import time
        import sys
        import os
```

1.12 Загружаем набор данных

Изображение в наборе данных CIFAR-10 цветные (то есть у них три канала: красный, зеленый и синий) содержат 10 классов, имеют размер 32x32 пикселя. Объявим эти переменные, так как дальше они будут часто использоваться.

```
In [2]: img_size = 32
        cropped_size = 24

        num_clsss = 10
        num_channels = 3
```

Для начала загрузим набор данных cifar-10:

```
In [3]: def download_and_extract():
            """
            Загружает и распаковывает набор данных cifar-10, если его еще нет в data_path
            """

            url="https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
            download_dir="data/CIFAR-10/"
            filename = url.split('/')[-1]
            file_path = os.path.join(download_dir, filename)

            if not os.path.exists(file_path):
                # создает директорию для загрузки, если она не существует
                if not os.path.exists(download_dir):
                    os.makedirs(download_dir)
```

```
                # скачивание файла с интернета
                file_path, _ = urllib.request.urlretrieve(url=url,
                                                          filename=file_path)

                if file_path.endswith(".zip"):
                    # распаковка zip
                    zipfile.ZipFile(file=file_path, mode="r").extractall(download_dir)
                elif file_path.endswith((".tar.gz", ".tgz")):
                    # распаковка tar
                    tarfile.open(name=file_path, mode="r:gz").extractall(download_dir)

                print("Done")
            else:
                print("Data has already been downloaded and unpacked")

        download_and_extract()
```

```
Data has already been downloaded and unpacked
```

и названия классов:

```
In [4]: def return_data(filename):
            file_path = os.path.join("data/CIFAR-10/",
                                     "cifar-10-batches-py/",
                                     filename)
            with open(file_path, mode='rb') as file:
                data = pickle.load(file, encoding='bytes')
            return data

        def load_class_names():
            raw = return_data(filename="batches.meta")[b'label_names']
            names = [x.decode('utf-8') for x in raw]
            return names

        class_names = load_class_names()
        class_names
```

```
Out[4]: ['airplane',
         'automobile',
         'bird',
         'cat',
         'deer',
         'dog',
         'frog',
         'horse',
         'ship',
         'truck']
```

Набор представляет 10 различных классов: самолет, автомобиль, птица, кошка, олень, собака, лягушка, лошадь, корабль и грузовик.

Загрузим тренировочный и тестовый наборы:

```
In [5]: def label_generation(class_numbers, num_clsss=None):
            """
            Генерирует метки классов в формате 1-мерного массива заполненым нулями
            с одним единичным элементом, чей индекс = номеру класса
            """

            if num_clsss is None:
                num_clsss = np.max(class_numbers) + 1

            return np.eye(num_clsss, dtype=float)[class_numbers]
```

```
In [6]: def convert_images(raw):
            """
            Конвертирует изображения с набора данных CIFAR-10
            в формат 4ех мерный массив [image_number, height, width, channel]
            """

            # конвертирует изображения из формата raw в точки float
            raw_float = np.array(raw, dtype=float) / 255.0

            # преобразует массив в 4-ех мерный
            images = raw_float.reshape([-1, num_channels, img_size, img_size])

            # изменяет порядок индексов
            images = images.transpose([0, 2, 3, 1])

            return images
```

```
In [7]: def load_data(filename):
            """
            Загружает файл данных из набора данных CIFAR-10
            и возвращает конвертированное изображение с номером класса для каждого изображения
            """

            # загружаем файл
            data = return_data(filename)
            raw_images = data[b'data']

            # загружаем номер класса
            cls = np.array(data[b'labels'])

            # конвертируем изображение
            images = convert_images(raw_images)

            return images, cls
```

```
In [8]: def load_training_data():
            """
            Загружает все картинки из CIFAR-10.
            Возвращает картинки, номера классов и матрицы принадлежностей.
            """

            # инициализируем массивы
            images = np.zeros(shape=[5 * 10000, img_size, img_size, num_channels], dtype=float)
            cls = np.zeros(shape=[5 * 10000], dtype=int)

            begin = 0

            for i in range(5):
                # загружаем изображение и номер класса
                images_batch, cls_batch = load_data(filename="data_batch_" + str(i + 1))
                num_images = len(images_batch)
                end = begin + num_images
                images[begin:end, :] = images_batch
                cls[begin:end] = cls_batch
                begin = end

            return images, cls, label_generation(class_numbers=cls, num_clsss=num_clsss)

In [9]: def load_test_data():
            """
            Загружает все картинки для тренировки из CIFAR-10.
            Возвращает картинки, номера классов и матрицы принадлежностей.
            """

            images, cls = load_data(filename="test_batch")
            return images, cls, label_generation(class_numbers=cls, num_clsss=num_clsss)

In [10]: images_train, cls_train, labels_train = load_training_data()
         images_test, cls_test, labels_test = load_test_data()
```

Набор данных CIFAR-10 теперь загружен и состоит из 60 000 изображений и связанных меток. Он разбит на 2 взаимоисключающих подмножества: тренировочный набор (50000 изображений) и тестовый набор (10000 изображений).

```
In [11]: print("Training-set:\t" + str(len(images_train)))
         print("Test-set:\t" + str(len(images_test)))
```

```
Training-set:        50000
Test-set:        10000
```

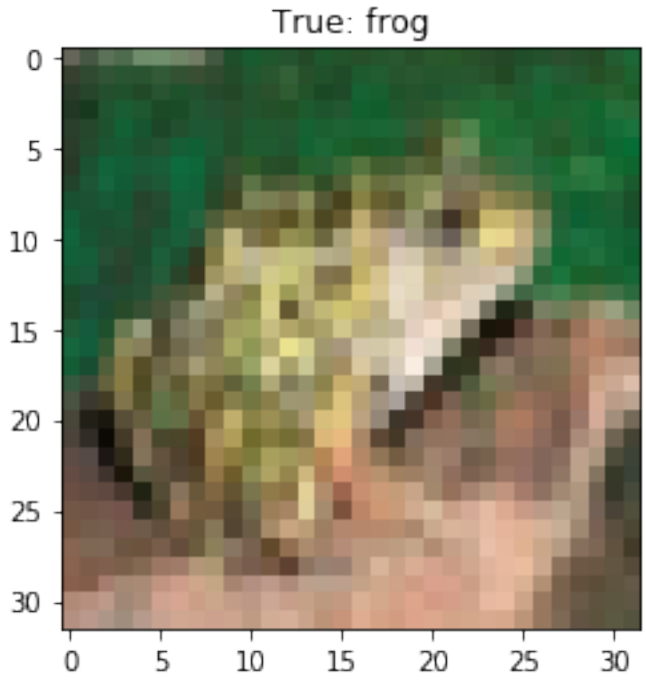
1.12.1 Функция для построения изображений

```
In [12]: def plot_image(image, cls_true, cls_pred=None):
             """
```

```
    Выводим изображение и его истинный класс и если есть - предсказанный
    """

    plt.imshow(image[0, :], interpolation='nearest')
    cls_true_name = class_names[cls_true[0]]

    # отображение названия истинного класса
    if cls_pred is None:
        xlabel = "True: {0}".format(cls_true_name)
    else:
        # отображение названия прогнозируемого класса
        xlabel = "True: {0}\nPred: {1}".format(cls_true_name, class_names[cls_pred[0]])

    plt.title(xlabel)
    plt.imshow(image[0, :, :, :], interpolation='nearest')
    plt.cls_true_name = class_names[cls_true[0]]
    plt.show()
```

Проверим, как будут отображаться данные:

```
In [13]: image_number = 512
         image = images_test[image_number:image_number+1]
         cls_true = cls_test[image_number:image_number+1]
         plot_image(image = image, cls_true=cls_true)
```



1.13 Граф тензора

Цель TensorFlow состоит в том, чтобы получить так называемый вычислительный граф, который может выполняться намного эффективнее, чем если бы те же вычисления выполнялись непосредственно в Python. TensorFlow может быть более эффективным, чем NumPy, потому что TensorFlow знает весь граф вычислений, который должен быть выполнен, а NumPy знает только вычисление одной математической операции за раз.

TensorFlow также может автоматически вычислять градиенты, которые необходимы для оптимизации переменных графа, чтобы сделать модель более эффективной. Это связано с тем, что граф представляет собой комбинацию простых математических выражений, поэтому градиент всего графа может быть рассчитан с использованием цепного правила для производных.

1.13.1 Placeholder variables

Placeholder – это просто переменная, предназначенная для определения позже. Эти переменные позволяют нам определять наши операции и строить граф вычислений без предварительного ввода данных. В терминологии TensorFlow через placeholder-ы мы передаем данные графу. Работает это следующим образом:

```
In [14]: x = tf.placeholder(tf.float32, shape=[None, img_size, img_size, num_channels], name='x'
```

Эта переменная будет хранить в себе истинные метки.

```
In [15]: lab_true = tf.placeholder(tf.float32, shape=[None, num_clsss], name='lab_true')
```

Мы могли бы также создать placeholder-переменную для номеров истинных классов, но легче вернуть индекс элемента равного 1.

```
In [16]: true_cls = tf.argmax(lab_true, dimension=1)
```

1.13.2 Функция для создания предварительной обработки

Эта функция добавляет узел к вычислительному графу для TensorFlow и предназначена для "расширения"обучающей выборки. Она рандомно искажает каждое изображение.

```
In [17]: def distort_image(img, training):
             """
             Принимает на вход изображение и значение типа bool, которое отслеживает
             изображение взято из тренировочной выборки или тестовой
             """

             if training:
                 # Рандомно обрезает изображение
                 img = tf.random_crop(img, size=[cropped_size, cropped_size, num_channels])

                 # Рандомно переворачивает изображение по горизонтали
                 img = tf.image.random_flip_left_right(img)

                 # Изменяет контраст, яркость, насыщенность и оттенок
                 img = tf.image.random_hue(img, max_delta=0.05)
```

```
            img = tf.image.random_contrast(img, lower=0.3, upper=1.0)
            img = tf.image.random_brightness(img, max_delta=0.2)
            img = tf.image.random_saturation(img, lower=0.0, upper=2.0)

            # Защита от переполнения
            img = tf.minimum(img, 1.0)
            img = tf.maximum(img, 0.0)
        else:
            # Обрезает изображение вокруг центра, так, чтобы по размеру оно было такое же,
            # как и изображение из тренировочной выборки
            img = tf.image.resize_image_with_crop_or_pad(img,
                                                         target_height = cropped_size,
                                                         target_width = cropped_size)

        return img

    def handling(images, training):
        images = tf.map_fn(lambda image: distort_image(image, training), images)
        return images

    distorted_images = handling(images=x, training=True)
```

1.13.3 Создание сверточной сети

С помощью функции `network` конструируем сеть. Мы будем использовать Pretty Tensor. В этой функции создается сеть с двумя сверточными слоями (`conv_1`, `conv_2`), с 64 ядрами размером = 5, двумя слоями субдискретизации (`max_pool`), которые уменьшают изображение в 2 раза, двумя полносвязными слоями (`fc_1` - 256 нейронов, `fc_2` - 128 нейронов) и слоем классификации (`softmax_classifier`).

In [18]:
```
def network(images, training):
    # складываем изображение в объект Pretty Tensor
    x_pretty = pt.wrap(images)

    if training:
        phase = pt.Phase.train
    else:
        phase = pt.Phase.infer

    # создаем сверточную нейронную сеть используя Pretty Tensor
    with pt.defaults_scope(activation_fn=tf.nn.relu, phase=phase):
        y_pred, loss = x_pretty.\
            conv2d(kernel=5, depth=64, name='conv_1', batch_normalize=True).\
            max_pool(kernel=2, stride=2).\
            conv2d(kernel=5, depth=64, name='conv_2').\
            max_pool(kernel=2, stride=2).\
            flatten().\
            fully_connected(size=256, name='fc_1').\
```

```
                    fully_connected(size=128, name='fc_2').\
                    softmax_classifier(num_classes=num_clsss, labels = lab_true)

        return y_pred, loss
```

Функция `create_network` создает нейронную сеть. Нейронная сеть объявлена как `network`.

```
In [19]: def create_network(training):
             with tf.variable_scope('network', reuse=not training):
                 images = handling(images=x, training=training)
                 y_pred, loss = network(images=images, training=training)
             return y_pred, loss
```

Переменная `global_step` будет отслеживать количество эпох оптимизации. В дальнейшем она нам понадобится для сохранения процесса обучения. `trainable=False` означает, что TensorFlow не будет пытаться оптимизировать эту переменную.

```
In [20]: global_step = tf.Variable(initial_value=0,
                                   name='global_step', trainable=False)
```

Создаем нейронную сеть, которая будет использоваться для обучения.

```
In [21]: _, loss = create_network(training=True)
```

Создаем оптимизатор, который минимизирует `loss`. Также передаем переменную `global_step` в оптимизатор, чтобы она была увеличена на единицу после каждой итерации.

`learning_rate` - это коэффициент скорости обучения, он выбирается в диапазоне от 0 до 1. Ноль указывать бессмысленно, поскольку в этом случае корректировка весов вообще производиться не будет. Выбор параметра противоречив. Большие значения (0,7 – 1) будут соответствовать большому значению шага коррекции. При этом алгоритм будет работать быстрее (т.е. для поиска минимума функции ошибки потребуется меньше шагов), однако может снизиться точность настройки на минимум, что потенциально увеличит ошибку обучения. Малые значения коэффициента (0,1 – 0,3) соответствуют меньшему шагу коррекции весов. При этом число шагов (или эпох), требуемое для поиска оптимума, как правило, увеличивается, но возрастает и точность настройки на минимум, что потенциально уменьшает ошибку обучения. В данном случае он равен `1e-4` или `0,0001`. Чаще всего на практике этот коэффициент выбирают экспериментально.

```
In [22]: optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(loss, global_step=globa
```

Теперь создайте нейронную сеть для этапа тестирования.

```
In [23]: y_pred, _ = create_network(training=False)
```

`y_pred` представляет собой массив с 10 элементами. В `pred_cls` будет записан индекс самого большого элемента в массиве.

```
In [24]: pred_cls = tf.argmax(y_pred, dimension=1)
```

Затем мы создаем вектор булевых строк, показывающий нам, является ли предсказанный класс равным истинному классу каждого изображения.

```
In [25]: correct_pred = tf.equal(pred_cls, true_cls)
```

Вектор `correct_pred` переводится из bool в float, так что: False становится 0, а True становится 1. Далее, расчитывается точность классификации, как среднее полученных чисел.

```
In [26]: acc = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

1.13.4 Saver

Объект типа Saver создается для того чтобы сохранить переменные нейронной сети, чтобы их можно было перезагрузить быстро, без необходимости снова тренировать сеть.

```
In [27]: saver = tf.train.Saver()
```

1.13.5 Веса и выходы с слоев

Мы использовали имена `conv_1` и `conv_2` для двух сверточных слоев. Pretty Tensor автоматически дает имена переменным, которые он создает для каждого слоя, поэтому мы можем получить веса для слоя с использованием имени.

```
In [28]: def get_weights(layer_name):
             with tf.variable_scope("network/" + layer_name, reuse=True):
                 variable = tf.get_variable('weights')

             return variable
```

Получаем переменные по каждому слою.

```
In [29]: weights_conv_1 = get_weights(layer_name='conv_1')
         weights_conv_2 = get_weights(layer_name='conv_2')
```

Аналогичным образом полуаем выходы со слоев. Функция немного отличается от выше приведенной, потому что нас интересует лишь последний тензор.

```
In [30]: def get_layer_output(layer_name):
             tensor_name = "network/" + layer_name + "/Relu:0"
             tensor = tf.get_default_graph().get_tensor_by_name(tensor_name)
             return tensor
```

Получаем выходы со сверточных слоев:

```
In [31]: output_conv_1 = get_layer_output(layer_name='conv_1')
         output_conv_2 = get_layer_output(layer_name='conv_2')
```

1.13.6 Создаем сессию TensorFlow

Объект Session инкапсулирует среду, в которой выполняются объекты операции, и оцениваются объекты тензора.

```
In [32]: session = tf.Session()
```

1.13.7 Контрольные точки

Обучение нейронной сети может занять много времени, особенно если у вас нет графического процессора. Поэтому мы сохраняем контрольные точки во время обучения, для того чтобы продолжить обучение в другое время (например, в ночное), а также для проведения анализа без необходимости тренировать нейронную сеть каждый раз, когда мы хотим ее использовать.

```
In [33]: # каталог для хранения контрольных точек
         save_dir = 'checkpoints/'

         # создаем каталог, если она еще не существует
         if not os.path.exists(save_dir):
             os.makedirs(save_dir)

         save_path = os.path.join(save_dir, 'cifar10_cnn')
```

Попробуйте восстановить контрольную точку:

```
In [34]: try:
             # нахождение контрольной точки
             last_chk_path = tf.train.latest_checkpoint(checkpoint_dir=save_dir)

             # пробуем восстановить последнюю кт
             saver.restore(session, save_path=last_chk_path)

             print("Restored checkpoint from:", last_chk_path)
         except:
             session.run(tf.global_variables_initializer())
```

```
INFO:tensorflow:Restoring parameters from checkpoints/cifar10_cnn-11110
Restored checkpoint from: checkpoints/cifar10_cnn-11110
```

1.13.8 Объем партии

В тренировочном наборе 50 000 изображений. Для вычисления градиента модели с использованием всех этих изображений требуется много времени. Поэтому мы используем только небольшую партию изображений на каждой итерации.

Это число можно варьировать. С увеличением этой переменной можно сокращать количество эпох и наоборот. При высоком значении - RAM будет перегружаться, а при маленьком будет высокая ошибка.

Эта функция выбирает случайный индекс из массива тренировочной выборки. И возвращает последующие batch_size картинок и их метки.

```
In [35]: batch_size = 100

         def random_batch():
             i = np.random.choice(len(images_train), size=batch_size, replace=False)
             return images_train[i, :, :, :], labels_train[i, :]
```

1.13.9 Оптимизация

Эта функция выполняет ряд эпох оптимизации. На каждой эпохе из набора тренировок выбирается новая партия данных, а затем TensorFlow выполняет оптимизатор с использованием этих данных. Прогресс печатается каждые 10 эпох. Контрольная точка сохраняется каждые 1000 эпох, а также после последней эпохи.

```
In [36]: def optimize(num_iterations):
             # используем таймер, для того, чтобы увидеть время
             start_time = time.time()

             for i in range(num_iterations):
                 x_batch, y_true_batch = random_batch()

                 feed_dict_train = {x: x_batch,
                                    lab_true: y_true_batch}

                 # запускаем оптимизатор
                 i_global, _ = session.run([global_step, optimizer],
                                           feed_dict=feed_dict_train)

                 # печатаем результат каждые 10 итераций
                 if (i_global % 10 == 0) or (i == num_iterations - 1):
                     # Срассчитываем accuracy на тренировочной выборке
                     batch_acc = session.run(acc,
                                             feed_dict=feed_dict_train)

                     # выводим статус
                     msg = "Epoch: {0:>6}, Accuracy: {1:>6.1%}"
                     print(msg.format(i_global, batch_acc))

                 # сохраняется последняя кт и после каждых 1000 эпох
                 if (i_global % 1000 == 0) or (i == num_iterations - 1):
                     saver.save(session,
                                save_path=save_path,
                                global_step=global_step)

                     print("Saved checkpoint.")

             # рассчитываем и выводим конечное время
             end_time = time.time()
             time_dif = end_time - start_time
             print("Time usage: " + str(timedelta(seconds=int(round(time_dif)))))
```

1.13.10 Построение сверточных весов

```
In [37]: def plot_conv_weights(weights, input_channel=0):
             w = session.run(weights)
```

```
    w_min = np.min(w)
    w_max = np.max(w)

    abs_max = max(abs(w_min), abs(w_max))

    num_filters = w.shape[3]
    num_grids = math.ceil(math.sqrt(num_filters))

    fig, axes = plt.subplots(num_grids, num_grids)

    for i, ax in enumerate(axes.flat):
        if i<num_filters:
            img = w[:, :, input_channel, i]
            ax.imshow(img, vmin=-abs_max, vmax=abs_max,
                      interpolation='nearest', cmap='seismic')

        ax.set_xticks([])
        ax.set_yticks([])

    return plt
```

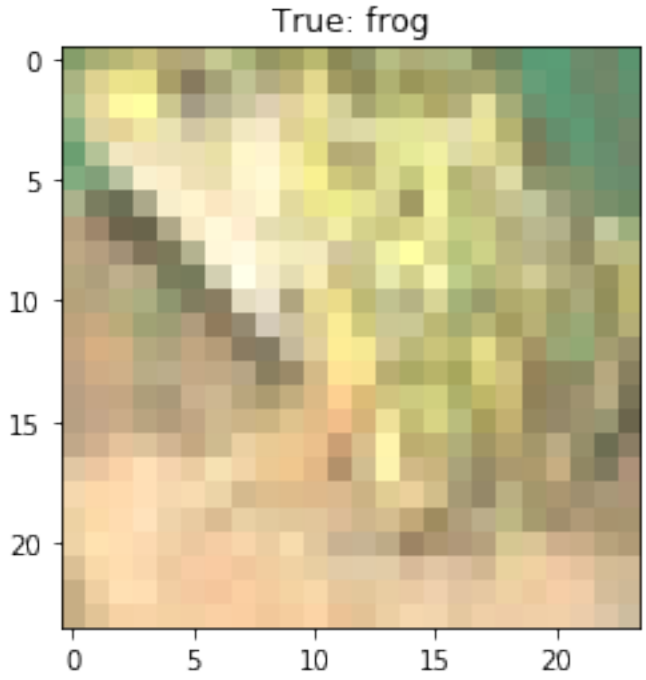
1.13.11 Построение выходов с светочных слоев

```
In [38]: def plot_layer_output(layer_output, image):
    feed_dict = {x: [image]}

    values = session.run(layer_output, feed_dict = feed_dict)

    values_min = np.min(values)
    values_max = np.max(values)

    num_images = values.shape[3]
    num_grids = math.ceil(math.sqrt(num_images))

    fig, axes = plt.subplots(num_grids, num_grids)

    for i, ax in enumerate(axes.flat):
        if i < num_images:
            img = values[0, :, :, i]

            ax.imshow(img, vmin = values_min, vmax = values_max,
                      interpolation = 'nearest', cmap = 'binary')

        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()
```

1.14 Пример искаженных изображений

```
In [39]: def plot_distorted_image(img, cls_true):
             img_2 = img[np.newaxis, :, :, :]
             feed_dict = {x: img_2}
             result = session.run(distorted_images, feed_dict=feed_dict)
             plot_image(image = result, cls_true = np.repeat(cls_true, 1))
```

```
In [40]: img = images_test[image_number, :, :, :]
         cls = cls_test[image_number]
         plot_distorted_image(img, cls)
```



1.15 Запускаем оптимизацию

От количества итераций и аппаратных показателей вашего компьютера зависит время обучения.

Поскольку мы сохраняем контрольные точки во время оптимизации и восстанавливаем последнюю контрольную точку при перезапуске кода, мы можем остановить и продолжить оптимизацию позже.

```
In [62]: optimize(num_iterations=1)
```

```
Epoch:  11120, Accuracy:  73.0%
Saved checkpoint.
Time usage: 0:00:01
```

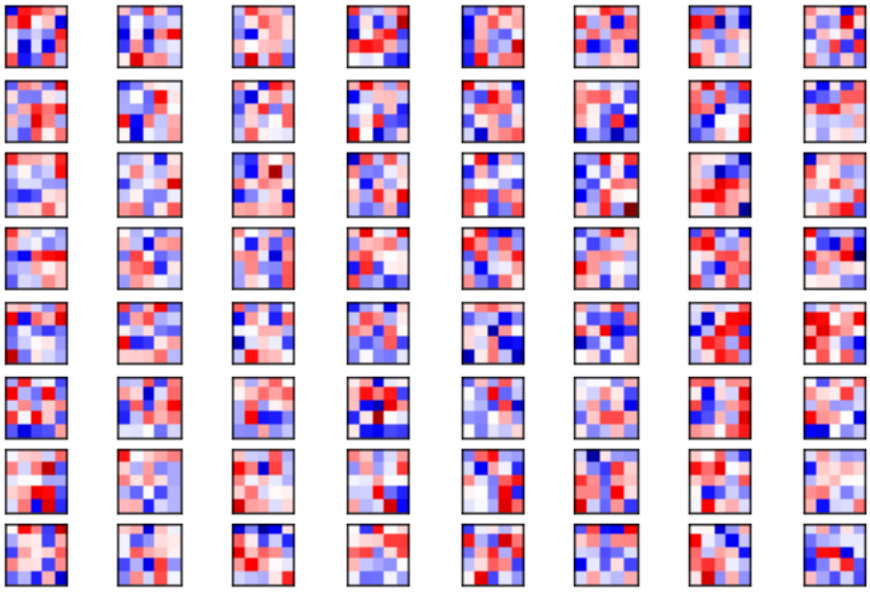
1.15.1 Веса

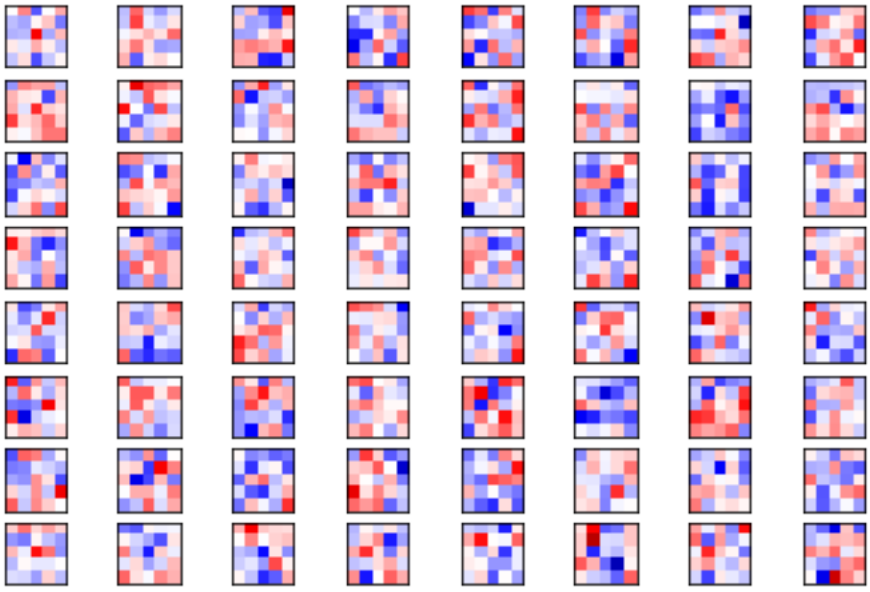
Посмотрим на веса с двух сверточных слев. Меняйте переменную `input_channel` (в диапазоне 0-2 для первого сверточного слоя и 0-63 для второго), чтобы посмотреть веса с других каналов.

Положительные веса - красные, отрицательные - синие.

In [42]:
```
plot_conv_weights(weights=weights_conv_1, input_channel=0)
plot_conv_weights(weights=weights_conv_2, input_channel=63)
```

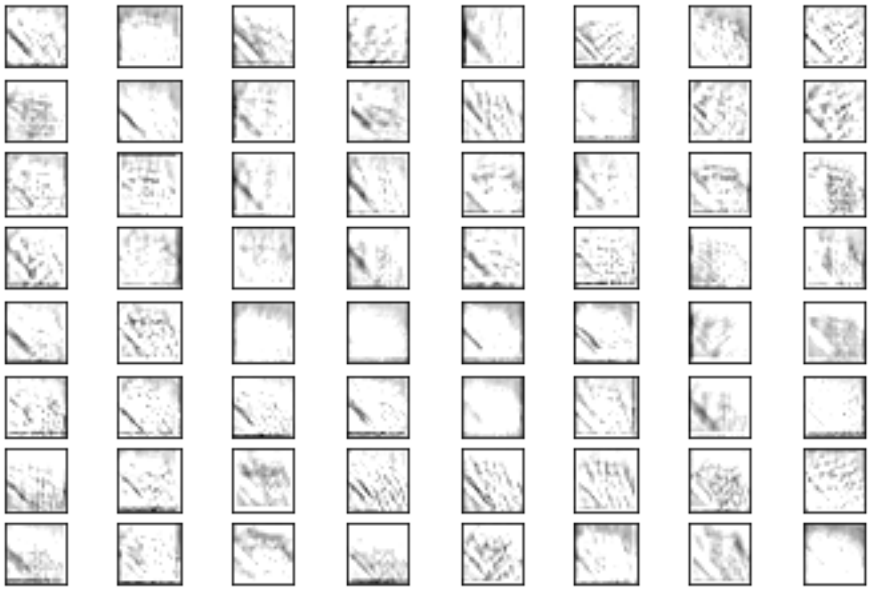
Out[42]:
```
<module 'matplotlib.pyplot' from '/anaconda3/lib/python3.6/site-packages/matplotlib/pyp
```

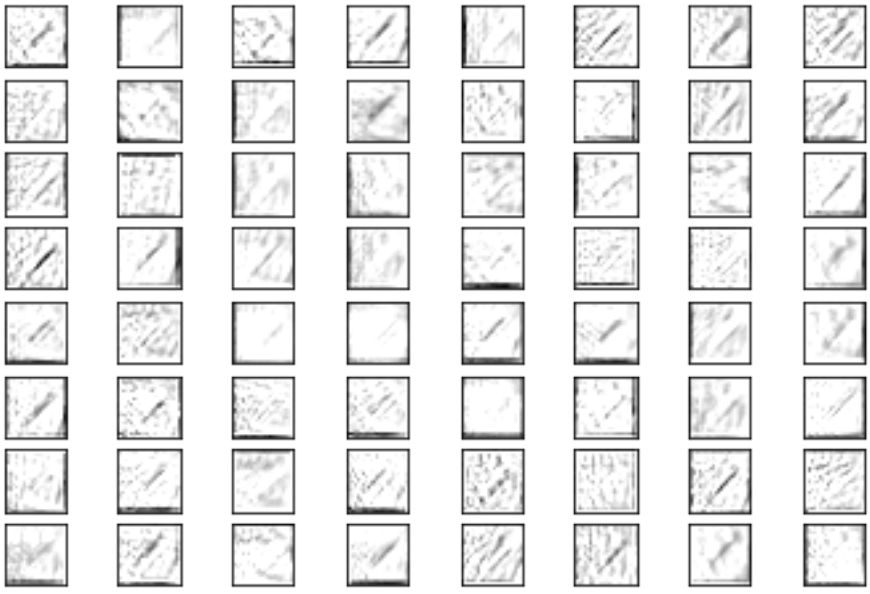




2 Выходы с сверточных слоев

In [43]:
```
plot_layer_output(output_conv_1, image = img)
plot_layer_output(output_conv_1, image = img)
```





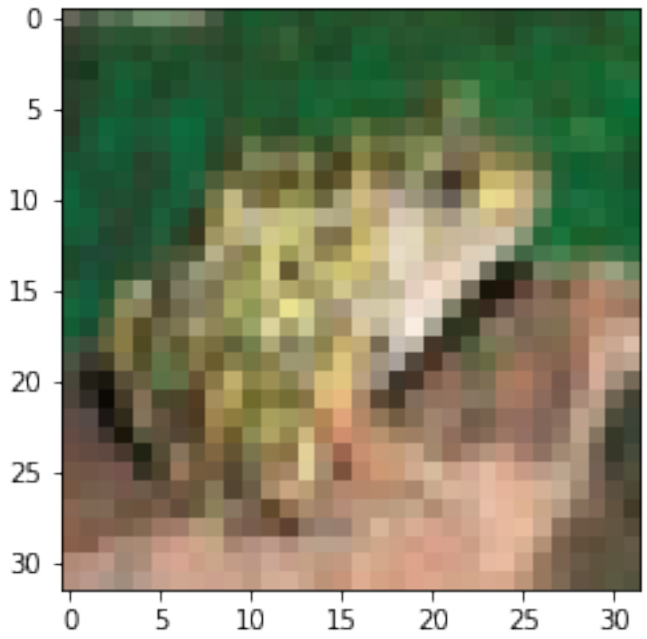
На данном графике сложно различить, какие же признаки сеть выявила после обучения. Но nhоэкспериментировав на разных классах можно заметить, что для машин это, например, колеса, прямые линии. Для лягушки - фактура.

2.1 Предсказания

Посмотрим на результаты предсказания:

```
In [53]: lbl_pred, cls_pred = session.run([y_pred, pred_cls],
                                          feed_dict={x: [img]})

         np.set_printoptions(precision=3, suppress=True)

         img = images_test[image_number, :, :, :]
         cls = cls_test[image_number]
         plt.imshow(img)
         print(lbl_pred[0])
```

```
[0.     0.     0.     0.     0.001  0.     0.999  0.     0.     0.  ]
```



Предсказание представляет из себя массив длинной 10. Каждый элемент массива показывает вероятность пренадлежности соответствующему классу.

В нашем случае на картинке лягушка, что соответствует седьмому классу и наше предсказание тоже считает, что это лягушка с вероятностью 0.999.

2.2 Закрытие TensorFlow сессии

По окончанию исспользования TensorFlow, обязательно закрывайте сессию.

```
In [45]: # session.close()
```

2.3 Заключение

В этом учебнике показано, как создать свернутую нейронную сеть для классификации изображений в наборе данных CIFAR-10. Точность классификации составляла около 80% на тестовом наборе.

2.4 Варианты

Создать свой классификатор на основе вышеприведенного на следующих наборах данных:

1. CIFAR-100 (взять любые 2-10 классов)
2. Fashion-MNIST
3. Linnaeus 5 dataset
4. Caltech-256
5. Caltech 101
6. CORe50
7.Cityscapes Dataset
8. HASYv2
9. MNIST