НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«Московский энергетический институт»
Кафедра математического и компьютерного моделирования

Исследование и разработка свёрточной нейронной сети для распознавания изображений

Выполнил: Сошников С. А.
Группа: А-16-19

Руководитель: Князев А. В.

Наиболее эффективным средством распознавания изображений на сегодняшний день является свёрточная нейронная сеть. Рассмотрим её устройство на примере моего проекта:

1) Свёрточный слой

```
class ConvolutionalLayer:
    def __init__(self, old_depth, old_height, old_width, fc, fs):       #
входные размеры, fc - кол-во фильтров, fs - длина и ширина фильтра
        self.old_depth = old_depth
        self.old_height = old_height
        self.old_width = old_width
        self.new_depth = fc
        self.new_height = self.old_height - fs + 1
        self.new_width = self.old_width - fs + 1
        self.filter_count = fc
        self.filter_size = fs
        self.filter_depth = old_depth
        self.W = np.zeros((fc, self.filter_depth, self.filter_size,
self.filter_size))      # веса ядер
        self.dW = np.zeros((fc, self.filter_depth, self.filter_size,
self.filter_size))     # градиенты весов ядер
        self.b = np.zeros((self.filter_count,))
# веса смещений
        self.db = np.zeros((self.filter_count,))
# градиенты весов смещений

        for f in range(self.filter_count):
            for i in range(self.filter_size):
                for j in range(self.filter_size):
                    for k in range(self.filter_depth):
                        (self.W[f])[k][i][j] = random.uniform(-0.1, 0.1)

            self.b[f] = 0.1



    def forward(self, input_tensor):
        output_tensor = np.zeros((self.new_depth, self.new_height,
self.new_width))
        for f in range(self.filter_count):
            for y in range(self.new_height):
                for x in range(self.new_width):
                    sum = self.b[f]

                    for i in range(self.filter_size):
                        for j in range(self.filter_size):
                            i0 = y + i
                            j0 = x + j
                            for k in range(self.filter_depth):
                                sum += input_tensor[k, i0, j0] *
(self.W[f])[k, i, j]
                    output_tensor[f, y, x] = sum
        return output_tensor

    def backward(self, input_tensor, deltas_tensor):
        for f in range(self.filter_count):
            for y in range(self.new_height):
                for x in range(self.new_width):
                    self.db[f] += deltas_tensor[f, y, x]
```

```
        for f in range(self.filter_count):
            for y in range(self.new_height):
                for x in range(self.new_width):
                    delta = deltas_tensor[f, y, x]
                    for i in range(self.filter_size):
                        for j in range(self.filter_size):
                            i0 = i + y
                            j0 = j + x
                            if i0 < 0 or i0 >= self.old_height or j0 < 0 or
j0 >= self.old_width:
                                continue
                            for k in range(self.filter_depth):
                                (self.dW[f])[k, i, j] += delta *
input_tensor[k, i0, j0]

        dX = np.zeros((self.old_depth, self.old_height, self.old_width))
        for y in range(self.old_height):
            for x in range(self.old_width):
                for k in range(self.filter_depth):
                    sum = 0
                    for i in range(self.filter_size):
                        for j in range(self.filter_size):
                            i0 = y + i - (self.filter_size - 1)
                            j0 = x + j - (self.filter_size - 1)
                            if i0 < 0 or i0 >= self.new_height or j0 < 0 or
j0 >= self.new_width:
                                continue
                            for f in range(self.filter_count):
                                sum += (self.W[f])[k, self.filter_size - 1 -
i, self.filter_size - 1 - j] * \
                                       deltas_tensor[f, i0, j0]
                    dX[k, y, x] = sum
        return dX

    def update_weights(self, learning_rate):
        for f in range(self.filter_count):
            for i in range(self.filter_size):
                for j in range(self.filter_size):
                    for k in range(self.filter_depth):
                        (self.W[f])[k, i, j] -= learning_rate *
(self.dW[f])[k, i, j]
                        (self.dW[f])[k, i, j] = 0
            self.b[f] -= learning_rate * self.db[f]
            self.db[f] = 0

    def activation_forward(self, tensor):
        result_tensor = np.zeros_like(tensor, dtype='float')
        for i in range(tensor.shape[1]):
            for j in range(tensor.shape[2]):
                for k in range(tensor.shape[0]):
                    if tensor[k, i, j] > 0:
                        result_tensor[k, i, j] = tensor[k, i, j]
                    else:
                        result_tensor[k, i, j] = 0
        return result_tensor

    def activation_backward(self, tensor, deltas):
        dX = np.zeros_like(tensor, dtype='float')
        for i in range(tensor.shape[1]):
            for j in range(tensor.shape[2]):
                for k in range(tensor.shape[0]):
                    dX[k, i, j] = deltas[k, i, j]
                    if tensor[k, i, j] > 0:
                        dX[k, i, j] *= 1
```

```
                    else:
                        dX[k, i, j] *= 0
        return dX
```

Конструктор принимает входные размеры тензора (глубина, высота, ширина), кол-во фильтров и размер фильтров (фильтры квадратные, N x N)
Выходная глубина тензора есть кол-во фильтров, выходные высота и ширина есть входные высота и ширина минус размер фильтра плюс 1.
Глубина фильтров всегда равна глубине входного тензора.
Также создаём массив тензоров, в каждый тензор будет записывать веса фильтров, очевидно, что длина этого массива равна кол-ву фильтров.
Аналогично создаются массивы для градиентов весов фильтра, весов смещения и градиентов весов смещения. Первоначально веса инициализируются случайными значениями от -0.1 до 0.1 (можно и другой диапазон малых чисел).

Далее метод forward отвечает за прямое распространение, в него передаётся входной тензор: сразу же формируется выходной тензор. Проходимся по каждой ячейке выходного тензора, для каждой ячейки создаём переменную sum, куда будем накапливать значения. Сперва прибавляем вес смещения, далее выполняем valid свёртку входного тензора с фильтром, результат также накапливаем в sum. Свёртку делаем по всей глубине тензора и фильтра. Результат (sum) записываем в соответствующую ячейку выходного тензора

Метод backward отвечает за обратное распространение, в него передаётся тензор, который был на входе, а также тензор ошибки, который пришёл от следующего слоя. Сперва вычисляем градиенты весов смещения, это очень просто: суммируем все дельты, которые относятся к данному весу смещения. Далее вычисляем градиенты весов фильтров, для этого делаем valid свёртку следующим образом: фиксируем определённую дельту, проходимся фильтром по входному тензору (как в случае прямого распространения), НО при прохождении в каждую ячейку тензора градиентов весов фильтров накапливаем результат перемножения этой (соответствующей) дельты на ячейку входного тензора. Проходимся при этом по всей глубине фильтра. Далее распространяем дельты от конца слоя к началу, для этого создаём тензор, соразмерный входному, и делаем full транспонированную свёртку. Создаём переменную sum, инициализированную нулём, куда накапливаем результат транспонированной свёртки. То есть веса фильтров поворачиваются на 180 градусов, и проделываем свёртку тензора дельт на перевёрнутые веса фильтров. Проходимся по всей глубине тензора и записываем результат.

Метод update_weights принимает гиперпараметр: скорость обучения (шаг спуска) и выполняем оптимизацию: обновление весов в соответствии с методом стохастического градиента: из каждой ячейки каждого фильтра отнимаем скорость обучения, помноженную на величину градиента.

Аналогично с весами смещения.

Далее следует метод activation_forward, принимающий входной тензор. Данный метод выполняет активацию результата свёртки. В моём случае в качестве функции активации выбрана функция ReLU (rectified linear unit). Формируется выходной тензор, соразмерный входному, далее каждый элемент входного тензора активируется по функции ReLU

Последний метод в данном классе – метод activation_backward, принимающий тензор, который был на входе, и тензор дельт со следующего слоя. Создаётся тензор, соразмерный входному, в его каждую ячейку записывается результат произведения дельты на производную функции активации с входным значением.

2) Пулинговый слой

```
class PoolingLayer:
    def __init__(self, old_depth, old_height, old_width, scale):
        self.old_depth = old_depth
        self.old_height = old_height
        self.old_width = old_width
        self.scale = scale
        self.new_depth = self.old_depth
        self.new_height = self.old_height // self.scale
        self.new_width = self.old_width // self.scale
        self.mask = np.zeros((self.old_depth, self.old_height,
self.old_width))

    def forward(self, input_tensor):
        output_tensor = np.zeros((self.new_depth, self.new_height,
self.new_width))
        for k in range(self.old_depth):
            for i in range(0, self.old_height, self.scale):
                for j in range(0, self.old_width, self.scale):
                    i_max = i
                    j_max = j
                    max = input_tensor[k, i, j]
                    for y in range(i, i + self.scale, 1):
                        for x in range(j, j + self.scale, 1):
                            value = input_tensor[k, y, x]
                            self.mask[k, y, x] = 0
                            if value > max:
                                max = value
                                i_max = y
                                j_max = x
                    output_tensor[k, i // self.scale, j // self.scale]
= max
                    self.mask[k, i_max, j_max] = 1
        return output_tensor

    def backward(self, deltas, input_tensor):
        dX = np.zeros_like(input_tensor, dtype='float')
        for k in range(self.old_depth):
            for i in range(self.old_height):
```

```
                for j in range(self.old_width):
                    dX[k, i, j] = deltas[k, i // self.scale, j //
self.scale] * self.mask[k, i, j]
        return dX
```

Конструктор данного слоя принимает размеры выходного тензора и коэффициент масштабирования (к.м.)
Выходная глубина равна входной, высота и ширина нацело делятся на к.м.
Также создаётся бинарная маска, соразмерная входному тензору.

Метод forward выполняет прямо распространение, принимая входной тензор. Сразу же формируется выходной тензор. Далее проходимся по глубине, для каждой карты тензора запускаем цикл по высоте, внутри – по ширине: с шагом, равным к.м., ищем максимальный элемент в заданном квадратике, его позицию и сохраняем эти данные. Максимум записываем в ячейку выходного тензора, а в бинарную маску записываем 1 в позицию максимума (в остальные места квадратика 0)

Метод backward выполняет обратное распространение, принимает входной тензор и тензор дельт. Создаём тензор, соразмерный входному. Проходимся по каждой его ячейке, в неё записываем результат произведения дельты на маску.

3) Полносвязный слой

```
class FullyConnectedLayer:
    def __init__(self, inputs, outputs):
        self.inputs = inputs
        self.outputs = outputs
        self.b = np.zeros((self.outputs,))
        self.db = np.zeros((self.outputs,))
        self.W = np.zeros((self.outputs, self.inputs))
        self.dW = np.zeros((self.outputs, self.inputs))
        self.df = np.zeros((self.outputs,))
        for i in range(self.outputs):
            for j in range(self.inputs):
                self.W[i, j] = random.uniform(-0.1, 0.1)
            self.b[i] = 0.1

    def activation(self, output):
        for i in range(self.outputs):
            output[i] = 1 / (1 + np.exp(-output[i]))
            self.df[i] = output[i] * (1 - output[i])

    def forward(self, inputed):
        output = np.zeros((self.outputs,))
        for i in range(self.outputs):
            sum = self.b[i]
            for j in range(self.inputs):
                sum += self.W[i, j] * inputed[j]
            output[i] = sum
        self.activation(output)
        return output

    def backward(self, deltas, inputed):
        for i in range(self.outputs):
```

```
            self.df[i] *= deltas[i]
        for i in range(self.outputs):
            for j in range(self.inputs):
                self.dW[i, j] = self.df[i] * inputed[j]
            self.db[i] = self.df[i]
        dX = np.zeros_like(inputed, dtype='float')
        for j in range(self.inputs):
            sum = 0
            for i in range(self.outputs):
                sum += self.W[i, j] * self.df[i]
            dX[j] = sum
        return dX

    def backward_final(self, outputed, inputed, need):
        global error
        deltas = np.zeros((self.outputs,))
        for i in range(self.outputs):
            deltas[i] = outputed[i] - need[i]
            error += (deltas[i] * deltas[i]) / 2
            self.df[i] *= deltas[i]
        for i in range(self.outputs):
            for j in range(self.inputs):
                self.dW[i, j] = self.df[i] * inputed[j]
            self.db[i] = self.df[i]
        dX = np.zeros_like(inputed, dtype='float')
        for j in range(self.inputs):
            sum = 0
            for i in range(self.outputs):
                sum += self.W[i, j] * self.df[i]
            dX[j] = sum
        return dX

    def update_weights(self, learning_rate):
        for i in range(self.outputs):
            for j in range(self.inputs):
                self.W[i, j] -= learning_rate * self.dW[i, j]
            self.b[i] -= learning_rate * self.db[i]
```

В конструктор данного слоя передаётся длина вектора входных данных и длина вектора выходных данных. Создаётся массив весов смещения, градиентов весов смещения и массив производных по кол-во выходных нейронов. Создаётся матрица весов, где строки соотв. выходным нейронам, столбцы – входным. Веса инициализируем малыми значениям (от -0.1 до 0.1) в моей программе

Метод activation проводит активацию выходного вектора нейронов, в моём случаем использована сигмоида. Также просчитываются производные функции активации для каждого нейрона.

Метод forward выполняет прямое распространение: формируется выходной вектор нейронов, в его каждую ячейку записывается взвешенная сумма входных нейронов плюс вес смещения. После просчёта всего выходного вектора, он передаётся на активацию.

Метод backward выполняет обратное распространение, принимая дельты следующего слоя и входной вектор. Для упрощения вычислений первым делом умножаем производные на дельты (i-ая производная на i-ую дельту), после чего запускаем цикл по выходным, а внутри – по входным нейронам, в нём i-ую производную умножаем на j-ый входной нейрон. Результат записываем в градиент i, j – го веса. На самом деле просто вспоминаем формулу расчёта градиента веса: вход синапса умножаем на дельту, помноженную на функцию активации. Аналогично считаем градиент веса смещения (у нейронов смещения выход всегда равен 1). Далее создаём вектор, соразмерный входному, в него распространяем дельты с выхода. В каждую ячейку созданного вектора записываем результата суммы произведений дельты, помноженной на производную, на вес.

Метод backward_final используется для выходного слоя, отличается тем, что в него передаётся имеющийся выход, эталонный выход, входной вектор. Используется глобальная переменная error как глобальная ошибка на эпохе. Создаются дельты как разность между реальным и эталонным выходом. В ошибку суммируется половина квадрата дельты, а далее – метод backward без изменений.

Метод update_weights выполняет обновление весов по стохастическому градиенту (аналогично свёрточному слою).

4) Предобработка входных данных

В моём случае использован датасет MNIST, но, понятное дело, данную СНС можно обучить на любой датасет. Выполняем пред обработку:

```
(train_X, train_Y), (test_X, test_Y) = mnist.load_data()
vector_train_Y = np.zeros((len(train_Y), 10))
vector_test_Y = np.zeros((len(test_Y), 10))
for i in range(len(train_Y)):
    vector_train_Y[i, train_Y[i]] = 1
for i in range(len(test_Y)):
    vector_test_Y[i, test_Y[i]] = 1
safe = np.zeros((len(train_X), 1, train_X.shape[1], train_X.shape[2]))
for i in range(len(train_X)):
    safe[i][0] = train_X[i]/255
train_X = safe.copy()
safe = np.zeros((len(test_X), 1, test_X.shape[1], test_X.shape[2]))
for i in range(len(test_X)):
    safe[i][0] = test_X[i]/255
test_X = safe.copy()
```

Массивы train_Y и test_Y преобразовываем в категориальный тип данных: массив из 10 элементов, 1 в нужном, 0 в остальном.
Преобразовываем train_X и test_X в массив тензоров, значения нормализуем путём деления пикселей на 255.

5) Создание архитектуры нейросети

```
layer1 = ConvolutionalLayer(1, 28, 28, 16, 3)
layer2 = ConvolutionalLayer(layer1.new_depth, layer1.new_height,
layer1.new_width, 16, 3)
layer3 = PoolingLayer(layer2.new_depth, layer2.new_height,
layer2.new_width, 2)
layer4 = ConvolutionalLayer(layer3.new_depth, layer3.new_height,
layer3.new_width, 32, 3)
layer5 = ConvolutionalLayer(layer4.new_depth, layer4.new_height,
layer4.new_width, 32, 3)
layer6 = PoolingLayer(layer5.new_depth, layer5.new_height,
layer5.new_width, 2)
layer7 = FullyConnectedLayer(layer6.new_depth * layer6.new_height *
layer6.new_width, 128)
layer8 = FullyConnectedLayer(layer7.outputs, 10)
```

В моём случае использовано 8 слоёв: свёрточный (16 фильтров 3x3), свёрточный (16 фильтров 3x3), пулинговый (к.м. 2), свёрточный (32 фильтра 3x3), свёрточный (32 фильтра 3x3), пулинговый (к.м. 2), полносвязный (128 нейронов), полносвязный (10 нейронов).

Теоретически данная нейросеть способна дать точность распознавания до 99,5% на датасете MNIST

6) Обучение нейросети

```
epoches = 10
epoch_error = np.zeros((epoches,))
for ep in range(1, epoches + 1):
    print("The epoch is ", ep)
    for t in range(len(train_X) - 59900):
        print(t)
        layer1_input = train_X[t].copy()
        instance = layer1.forward(train_X[t])
        layer1_activat_input = instance.copy()
        instance = layer1.activation_forward(instance)
        layer2_input = instance.copy()
        instance = layer2.forward(instance)
        layer2_activat_input = instance.copy()
        instance = layer2.activation_forward(instance)
        layer3_input = instance.copy()
        instance = layer3.forward(instance)
        layer4_input = instance.copy()
        instance = layer4.forward(instance)
        layer4_activat_input = instance.copy()
        instance = layer4.activation_forward(instance)
        layer5_input = instance.copy()
        instance = layer5.forward(instance)
        layer5_activat_input = instance.copy()
        instance = layer5.activation_forward(instance)
        layer6_input = instance.copy()
        instance = layer6.forward(instance)

        instance_array = np.zeros((instance.shape[0] * instance.shape[1] *
instance.shape[2],))
        index = -1
        for k in range(instance.shape[0]):
            for i in range(instance.shape[1]):
                for j in range(instance.shape[2]):
                    index += 1
```

```
                    instance_array[index] = instance[k, i, j]

        layer7_input = instance_array.copy()
        instance_array = layer7.forward(instance_array)
        layer8_input = instance_array.copy()
        instance_array = layer8.forward(instance_array)
        print(instance_array)
        error = 0
        my_deltas = layer8.backward_final(instance_array, layer8_input,
vector_train_Y[t])
        epoch_error[ep - 1] += error
        layer8.update_weights(0.4)
        my_deltas = layer7.backward(my_deltas, layer7_input)
        layer7.update_weights(0.4)

        deltas = np.zeros_like(instance, dtype='float')
        index = -1
        for k in range(instance.shape[0]):
            for i in range(instance.shape[1]):
                for j in range(instance.shape[2]):
                    index += 1
                    deltas[k, i, j] = my_deltas[index]

        deltas = layer6.backward(deltas, layer6_input)
        deltas = layer5.activation_backward(layer5_activat_input, deltas)
        deltas = layer5.backward(layer5_input, deltas)
        layer5.update_weights(0.4)
        deltas = layer4.activation_backward(layer4_activat_input, deltas)
        deltas = layer4.backward(layer4_input, deltas)
        layer4.update_weights(0.4)
        deltas = layer3.backward(deltas, layer3_input)
        deltas = layer2.activation_backward(layer2_activat_input, deltas)
        deltas = layer2.backward(layer2_input, deltas)
        layer2.update_weights(0.4)
        deltas = layer1.activation_backward(layer1_activat_input, deltas)
        deltas = layer1.backward(layer1_input, deltas)
        layer1.update_weights(0.4)
        print('MISTAKE')
        print(epoch_error)
```

Задаём 10 эпох, создаём массив ошибок длиной кол-ва эпох, в каждую его ячейку будем записывать ошибку на эпохе. Запускаем цикл по числу эпох, в каждом цикле обучаем нейросеть по 100 первым прецедентам из датасета, скорость обучения 0.4.
В результате получаем следующие показатели ошибки

```
MISTAKE
[49.01583926 46.82598887 46.34869897 45.91090231 43.5831155  37.59395628
 24.85556116 19.70024379 16.36753937 13.0701178 ]
```

Видим, что за 10 эпох по 100 прецедентам ошибка нейросети уменьшилась более, чем в три раза, что очень хороший результат.