# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Классификация изображений является одной из основных задач современного компьютерного зрения. С появлением вычислительных устройств человечество начало искать программные решения для задач которые ранее выполнялись исключительно человеком. Примерами таких задач являются распознавание символов, медицинская диагностика и анализ рентгеновских снимков, сортировка брака на производстве, распознавание лиц и многие другие.

С появлением глубокого обучения многие перечисленные проблемы к текущему моменту были решены. В частности, наиболее эффективными оказались так называемые ”сверточные” нейронные сети, также известные как CNN (Convolutional Neural Network), речь о которых пойдет в данной работе. Подробнее о понятии свертки рассматривается в теоретическом разделе.

Идея сверточных сетей имеет как математические, так и биологические основания. Основная идея CNN пришла из нейрофизиологии. Дэвидом Хьюбелом было обнаружено, что нейроны первичной зрительной коры реагируют на локальные шаблоны или признаки (например, линии под определенным углом), а нейроны более глубоких слоев на основе этих признаков выделяют все более сложные структуры [1]. Этим также обосновывается использование в CNN не только сверточных, но и других слоев, которые обобщают работу сверточных.

Идея того, как именно в сверточных сетях организована ”локальность”, пришла из математики. До появления глубокого обучения при работе с изображениями уже использовали так называемые фильтры – матрицы небольшого размера, как бы ”скользящие” по изображению. С помощью правильно подобранного фильтра, например, можно выделять границы различных объектов или иные свойства. Сверточные сети подбирают эти фильтры вместо человека, используя математический аппарат глубокого обучения. Так называемое

”скольжение” фильтра по изображению можно описать математической операцией свертки, от которой CNN и получили свое называние.

Сверточные нейронные сети решили проблему огромного числа параметров, которые используются полносвязными нейронными сетями. При высоком разрешении входно изображения, например $512 \times 512$ (по современным меркам даже такое разрешение уже не является большим), требует для каждого нейрона первого слоя иметь $512 \times 512 = 262144$ параметров, что невероятно много даже для современных компьютеров.

Несмотря на появление новых алгоритмов глубокого обучения, с помощью которых также решается задача классификации, сверточные сети все еще остаются актуальными. Помимо малого числа параметров по сравнению с полносвязными нейронными сетями, сверточные сети лучше поддаются анализу и интерпретации работы модели. На текущий момент разработаны методы, позволяющие визуализировать выделяемые сетями признаки [2].

Сама же задача классификации изображений остается актуальной и по сей день. Сегодня ее используют при создании автономных автомобилей, в социальных сетях и рекомендательных системах, робототехнике.

## Постановка задачи

Целью работы является изучение методов разработки, обучения и тестирования сверточной нейронной сети, решающей задачу классификации изображений из датасета https://www.kaggle.com/datasets/alessiocorrado99/animals10.

# 1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ СВЕРТОЧНЫХ СЕТЕЙ

## 1.1 Общая теория глубокого обучения

Прежде чем говорить о сверточных нейронных сетях, стоит рассмотреть ключевые понятия в теории глубокого обучения. Самой базовой моделью глубокого обучения называется так называемый **многослойный перцептрон**, или **MLP**.

Основная идея данной модели заключается в построении отдельных слоев линейных моделей, называемых **перцептронами**, связанных со всеми моделями следующего слоя некой нелинейной функцией, называемой **функцией активации**. На рисунке 1 изображена примерная схема MLP, где каждая стрелка означает вход с применением функции активации.

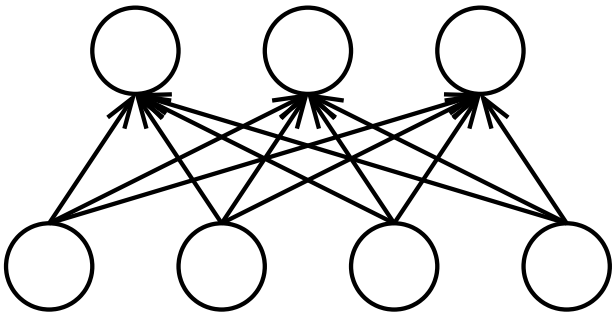

**Рисунок 1** – Схематичное изображение MLP

Используют различные функции активации. В конце XX века чаще использовали сигмоидные функции $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$ и $\sigma(x) = \frac{1}{1+e^{-1x}}$. Изображение этих функций представлено на рисунке 2.

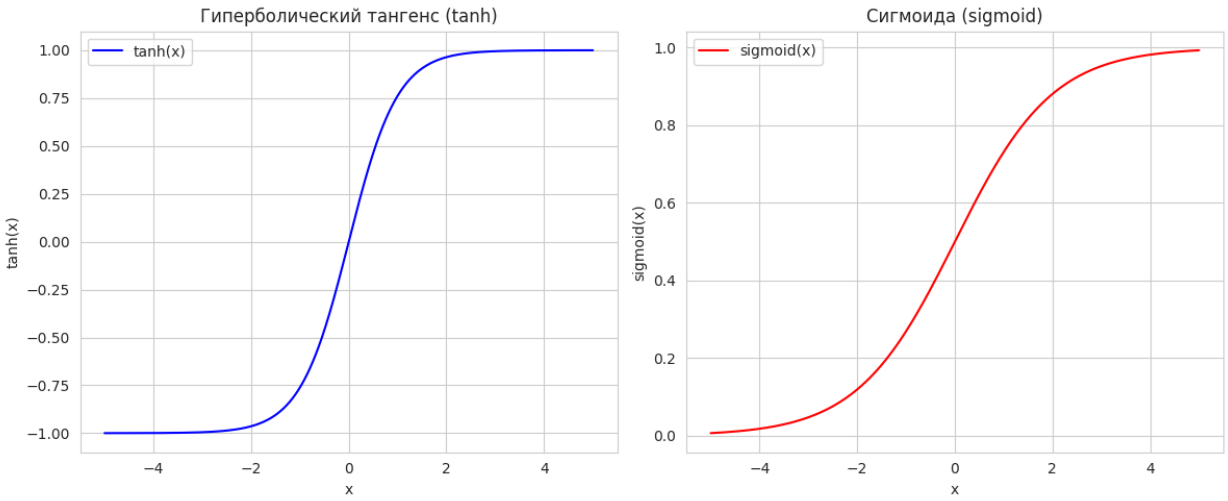


**Рисунок 2** – Сигмоидные функции активации

Современные нейросетевые решения используют функцию активации $\mathrm{ReLU}(x) = \frac{x+|x|}{2} = \max(0, x)$ (rectified linear unit). Ее изображение представлено на рисунке 3.

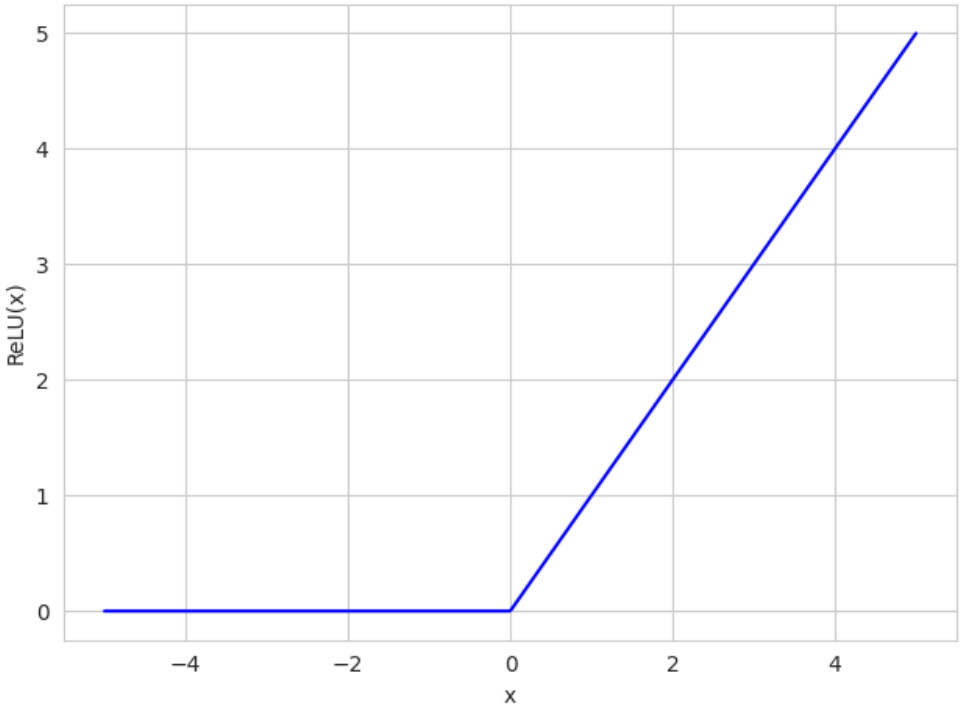


**Рисунок 3** – Функция активации ReLU

Популярность функции ReLU связана с проблемой затухающих градиентов [1]. Вне окрестности нуля сигмоидные функции имеют очень малые значения производной, что негативно сказывается на обучении модели.

Теперь рассмотрим, каким образом будет выглядеть выходная функция, преобразующая исходный вектор $x$ в вектор $z$ последнего слоя. Под слоем теперь будем подразумевать произвольную дифференцируемую (за исключением счетного числа точек) функцию многих переменных $x$ и $w$. Если модель имеет $l$ слоев $f_l, \ldots, f_1$ с параметрами $w_l, \ldots, w_1$ на каждом слое, то выходную функцию можно представить в виде:

$$z = f_l(w_l, f_{l-1}(\ldots f_1(w_1, x))\ldots)) = \hat{f}(x, \omega_1, \ldots, \omega_l) \tag{1}$$

Таким образом, нейросеть представляет собой композицию параметрических функций [3]. Поставленную задачу можно в общем случае описать как аппроксимацию некой функции $z = f(x)$ с помощью композиции параметрических функций. Для идентификации неизвестных параметров $w_i, i = 1, \ldots, l$ необходим набор пар $(x_i, y_i)$, где $y = f(x)$ и $i = 1, \ldots, N$, $N$ – размер обучаемого набора.

Задачу подбора параметров модели решают с помощью введения скалярной функции многих переменных, называемой **функцией потерь**, которая зависит от точных значений выходных переменных $y = f(x)$ и выходных переменных модели $\hat{y} = \hat{f}(x, \omega_1, \ldots, \omega_l)$. Данная функция характеризует, насколько точно модель предсказывает истинные значения $y$. Решая задачу минимизации (максимизации) данной функции, получаем некоторую оценку истинных параметров модели.

В задаче классификации используют функцию потерь, называемую **перекрестной энтропией**, или **crossentropy loss** [3]. Как правило, выходной слой классификационной модели интерпретируется как вектор вероятностей классификации каждого класса. Математически перекрестная энтропия описывается так:

$$H = \frac{1}{N} \sum_{i=1}^{N} \log(P(y_i = \hat{y}_i \mid x_i, w)), \tag{2}$$

где $P(y = \hat{y} \mid x, w)$ – вероятность значения $y_i$, предсказанная моделью.

Для минимизации функции потерь на практике используют алгоритм градиентного спуска или его модификации. Для оптимального вычисления градиентов в нейрнонных сетях был разработан специальный алгоритм **backpropagation** [4].

## 1.2 Сверточные нейронные сети

Определим операцию дискретной свертки двух последовательностей $x = \{x_i\},\ y = \{y_i\}$:

$$(x * y)(t) = \sum_i x_i \cdot y_{t-i} \tag{3}$$

Аналогично можно определить двумерную свертку для двумерных последовательностей $X = \{X_{i,j}\},\ Y = \{Y_{i,j}\}$:

$$(X * Y)(t, s) = \sum_{i,j} X_{i,j} \cdot Y_{t-i,s-j} \tag{4}$$

Если же двумерные векторы конечны, то есть их можно представить в виде матриц, операцию свертки интерпретируют как наложение окна $Y$ с симметрией относительно побочной диагонали и сдвигом на $(t, s)$ на матрицу $X$, с последующим суммированием (рисунок 4). Таким образом, свертка в точке $(t, s)$ максимальна, когда матрица $X$ равна повернутой матрице $Y$ со сдвигом на $(t, s)$.

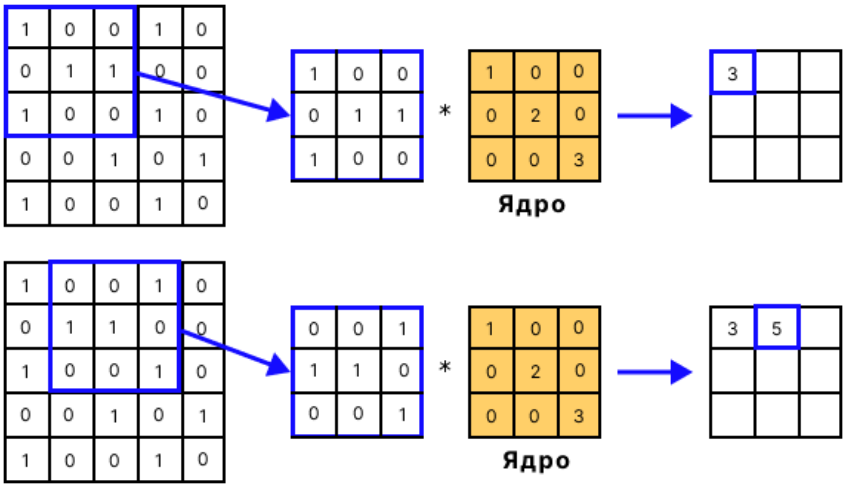


**Рисунок 4** – Визуализация операции свертки с ”перевернутым” ядром

В связи с неудобством рассмотрения повернутой матрицы $Y$ под сверткой часто подразумевают несколько иную операцию, называемой **взаимной корреляцией** [1]:

$$(X \otimes Y)(t, s) = \sum_{i,j} X_{i,j} \cdot Y_{i+t,j+s} \tag{5}$$

Такая операция теряет свойство симметричности, однако удобнее в своей интерпретируемости. В дальнейшем под двумерной сверткой будет подразумеваться именно взаимная корреляция.

Одним из способов анализировать изображения, который использовали до появления современных алгоритмов машинного обучения, является метод матричных фильтров. Рассмотрим изображение в черно-белом формате разрешения $n \times m$:

$$X = \begin{pmatrix} X_{11} & \dots & X_{1m} \\ \vdots & \dots & \vdots \\ X_{n1} & \dots & X_{nm} \end{pmatrix}$$

Фильтром размера $k \times l$, где $k < n,\ l < m$ назовем матрицу:

$$K = \begin{pmatrix} K_{11} & \dots & X_{1l} \\ \vdots & \dots & \vdots \\ K_{k1} & \dots & K_{kl} \end{pmatrix}$$

Метод фильтров подразумевает вычисление свертки матрицы и фильтра. Подбирая подходящий фильтр и смотря на величину свертки, можно производить анализ изображения. Например, если фильтр характеризует ”границу” некоторого объекта, по значению свертки можно выделить эту границу.

**Сверточным слоем** в нейронной сети будем называть слой, преобразующий входную переменную $x$ в свертку этой переменной с **ядром** слоя $k$:

$$z = k \otimes x$$

Использование сверточных слоев в случае, когда входной переменной являет-

ся матрица (например, изображение), обосновывается методом фильтров, который успешно применяли в прошлом.

Сверточный слой отличается от полносвязного тем, что нейроны следующего слоя связаны с предыдущими локально. Локальность в случае одномерных слоев представлена на рисунке 5.

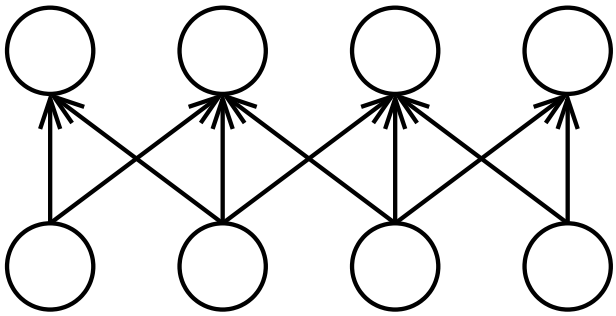

**Рисунок 5** – Локальность сверточных слоев

На практике сверточные слои дополнительно модифицируют. Помимо размера ядра, есть параметры ”padding” и ”stride”. Параметр padding дополняет входную матрицу нулями по краям, чтобы выходной вектор был другого размера. Параметр stride влияет на то, сколько сверток ”пропускается”. Пример того, как выглядит сеть при stride $= 2$, представлен на рисунке 6.

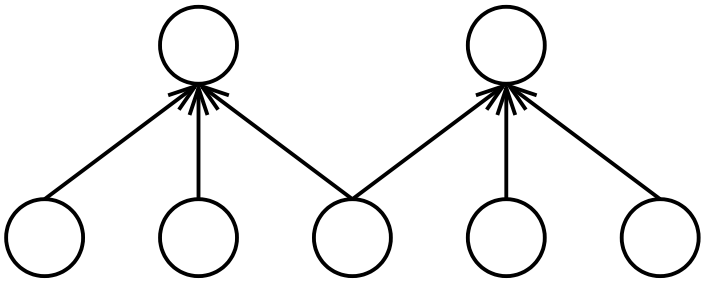

**Рисунок 6** – Вид сверточного слоя при stride $= 2$

Помимо сверточных слоев при построении CNN важно добавлять ”pooling”

слои. В отличие от сверточных ядер, ядро pooling слоя является непараметрической функцией от ”окна”. На практике используют ”max pooling” (максимум) и ”average pooling” (среднее) слои. Параметры ”padding” и ”stride” для pooling слоев работают также, как и для сверточных.

При построении моделей на каждом слое подбирается несколько фильтров, у каждого из которых есть своя матрица на выходе. Выходную матрицу одной свертки называют **каналом**. В общем случае на входе сверточного слоя находится тензор 3-го порядка $X_{ijk}$, где $i$ – номер канала, $j$ и $k$ – пространственные координаты. Ядро каждого канала также становится трехмерным. Общая формула свертки:

$$\text{conv}(X, K)_{i'j'k'} = \sum_{i,j,k} X_{i,j+j',k+k'} K_{i'ijk}, \tag{6}$$

где в ядре $K_{i'ijk}$ индекс $j'$ определяет номер выходного канала, $j$ – номер входного канала.

## 1.3 Нормализация слоев

Между сверточными, pooling и полносвязными слоями можно ставить вспомогательные слои, выполняющие различные функции. При решении поставленной задачи использовались 3 подобных слоя, а именно: ”batch” нормализация, ”local response” нормализация, и ”dropout” нормализация.

**Batch** нормализация используется между слоем и его функцией активации. Его используют в случае, когда на вход модели подается параллельно сразу M входных изображений, называемых батчем размера M. Сначала происходит нормализация ”по батчу”, то есть вычисление среднего и дисперсии:

$$\begin{aligned} \mu_{jk} &= \frac{1}{M} \sum_{i=1}^{M} X_{ijk} \\ \sigma_{jk} &= \frac{1}{M} \sum_{i=1}^{M} (X_{ijk} - \mu_{jk})^2 \end{aligned} \tag{7}$$

Затем входной вектор нормализуется:

$$\hat{X}_{ijk} = \frac{X_{ijk} - \mu_{jk}}{\sqrt{\sigma_{jk}^2 + \varepsilon}} \tag{8}$$

Последним этапом batch нормализации является масштабирование:

$$Y_{ijk} = \gamma_{jk}\hat{X}_{ijk} + \beta_{jk}, \tag{9}$$

где $\gamma_{jk}$ и $\beta_{jk}$ являются обучаемыми параметрами. Засчет масштабирования в конце модель может ”отменить” проделанную нормализацию, если это необходимо [5].

**Local response** нормализация усредняет выход нейронного слоя по каналам [7]:

$$Y_c = X_c \left( k + \frac{\alpha}{n} \sum_{c'=\max(0,c-\frac{n}{2})}^{\min(N-1,c+\frac{n}{2})} X_{c'} \right)^{-\beta}, \tag{10}$$

где $X_c$ – канал, $k$, $n$, $\alpha$, $\beta$ – необучаемые параметры.

**Dropout** нормализация с заданной вероятностью ”отключает” выход каждого нейрона в слое, после которого поставлен dropout. Отключение нейронов происходит только во время обучения модели. Данный слой помогает уменьшить переобучение, поскольку на каждой итерации для предсказания используется меньшее число параметров [6].

## 1.4 Визуализация работы CNN

Глубокие сверточные нейронные сети обучаются определять все более сложные признаки на каждом слое [1]. Можно предположить, что последний сверточный слой обладает балансом между сложностью признаков и их интерпретируемостью, в отличие от первых сверточных слоев (слабые признаки) и полносвязных слоев модели (слишном сложные признаки).

Исходя из этого предположения был предложен градиентный алгоритм визуализации Grad-CAM [2]. Пусть модель решает задачу классификации, $y_c$

– выход модели по отношению к $i$ классу, $A^k$ – выход $k$-го канала последнего сверточного слоя. Для получения тепловой карты, на которой будет изображена важность разных частей изображения, необходимо вычислить важность $k$-го канала:

$$\alpha_k^c = \frac{1}{Z} \sum_{i,j} \frac{\partial y^c}{\partial A_{ij}^k} \tag{11}$$

Затем вычисляется тепловая карта малого разрешения:

$$L_{\text{Grad-CAM}}^c = \text{ReLU}(\sum_k \alpha_k^c A^k) \tag{12}$$

Функция ReLU ”фильтрует” признаки, которые отрицательно влияют на выход $y^c$. Полученная тепловая карта интерполируется до разрешения исходного изображения с помощью известных методов [2].

# 2. РЕАЛИЗАЦИЯ И АНАЛИЗ ЭФФЕКТИВНОСТИ МОДЕЛЕЙ

В работе были изучены и реализованы две архитектуры сверточных нейронных сетей: AlexNet [7] и VGGNet [8]. Были произведены модификации данных моделей под решаемую задачу. Для реализации использовалась библиотека глубокого обучения **pytorch**. Помимо функции потерь оценивалась точность предсказаний.

Для обучения представленных в работе моделей использовался компьютер со следующими характеристиками:

— Процессор Intel Core i5 12400h (6 ядер)

— 32 ГБ оперативной памяти

— Видеокарта AMD Radeon RX6700XT (12 ГБ видеопамяти)

## 2.1 Предобработка данных

Исходный датасет состоит из 26179 изображений. На рисунке 7 представлена 2D гистограмма распределения разрешения изображений.

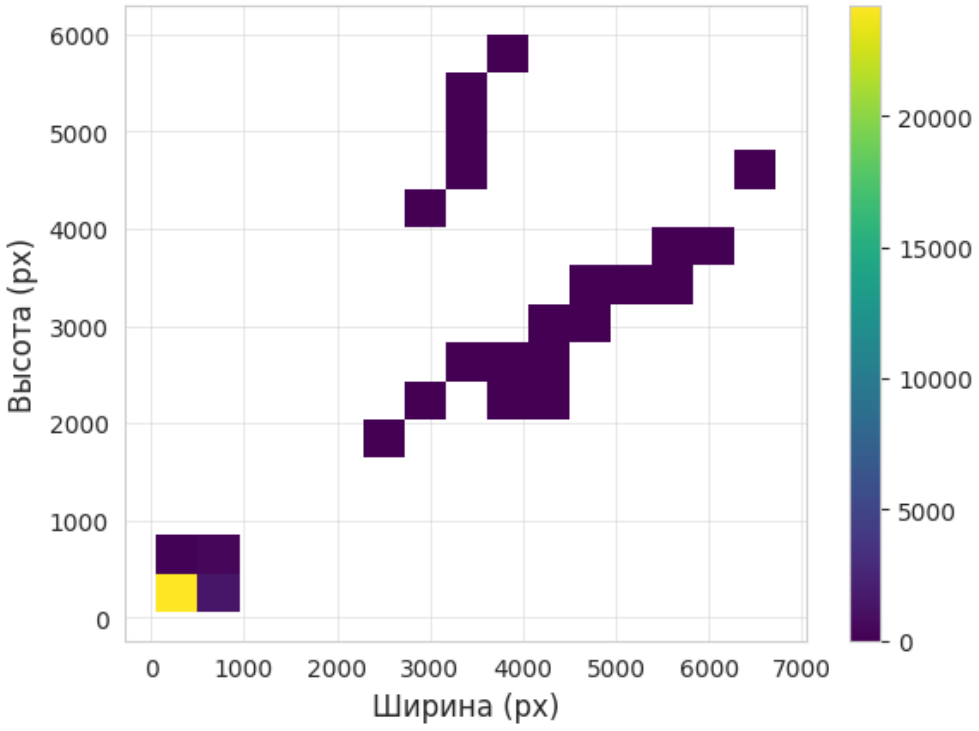


**Рисунок 7** – Распределение разрешения изображений

Исходя из вида распределения разрешений, было решено предварительно привести все изображения к разрешению $256 \times 256$ пикселей.

Производилась **аугментация изображений**: обрезка со случайным центром до разрешения $224 \times 224$, случайная горизонтальная симметрия, случайный поворот на угол до $20°$. Аугментация помогает модели верно предсказывать класс перевернутых изображений, даже если в обучаемом наборе данных животные были расположены примерно под одним углом.

При обучении моделей учитывался баланс классов. Он учитывался как при разбиении данных на тестовые и тренировочные, так и при рассчете перекрестной энтропии. Распределение данных по классам представлено на рисунке 8

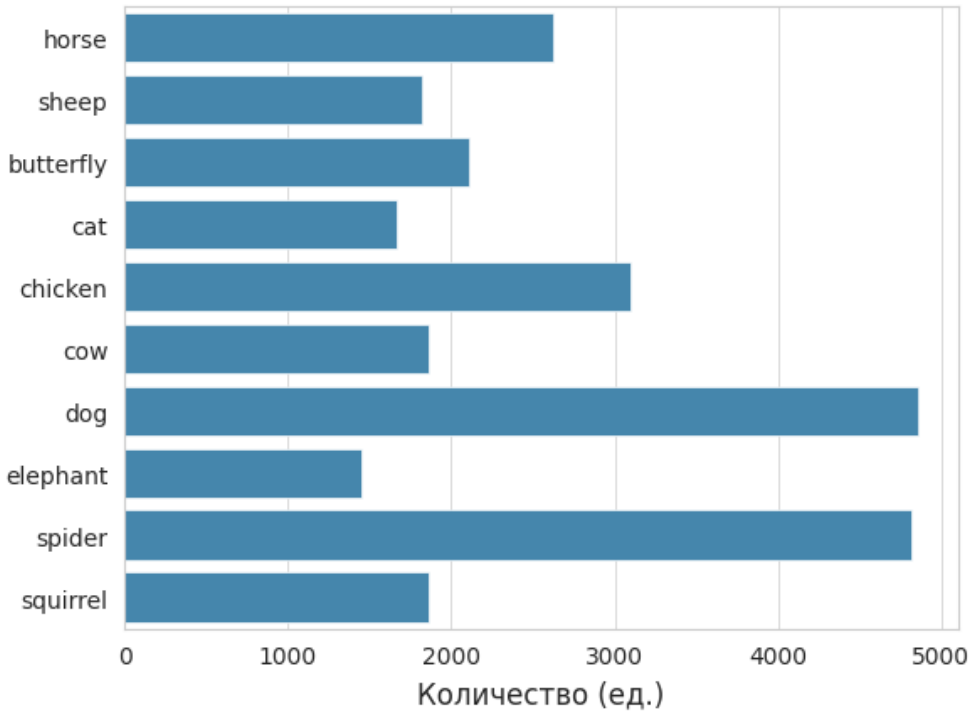


**Рисунок 8** – Распределение данных по классам

## 2.2 Архитектура AlexNet

Архитектура AlexNet основана на идее постепенного уменьшения размера ядра сверток. На первом сверточном слое используется ядра размера $11 \times 11$, затем ядро $5 \times 5$ и три свертки с ядром $3 \times 3$. Большие ядра на первых сверточных

слоях позволяют выделять большие объекты сразу. Схема модели представлена на рисунке 9.

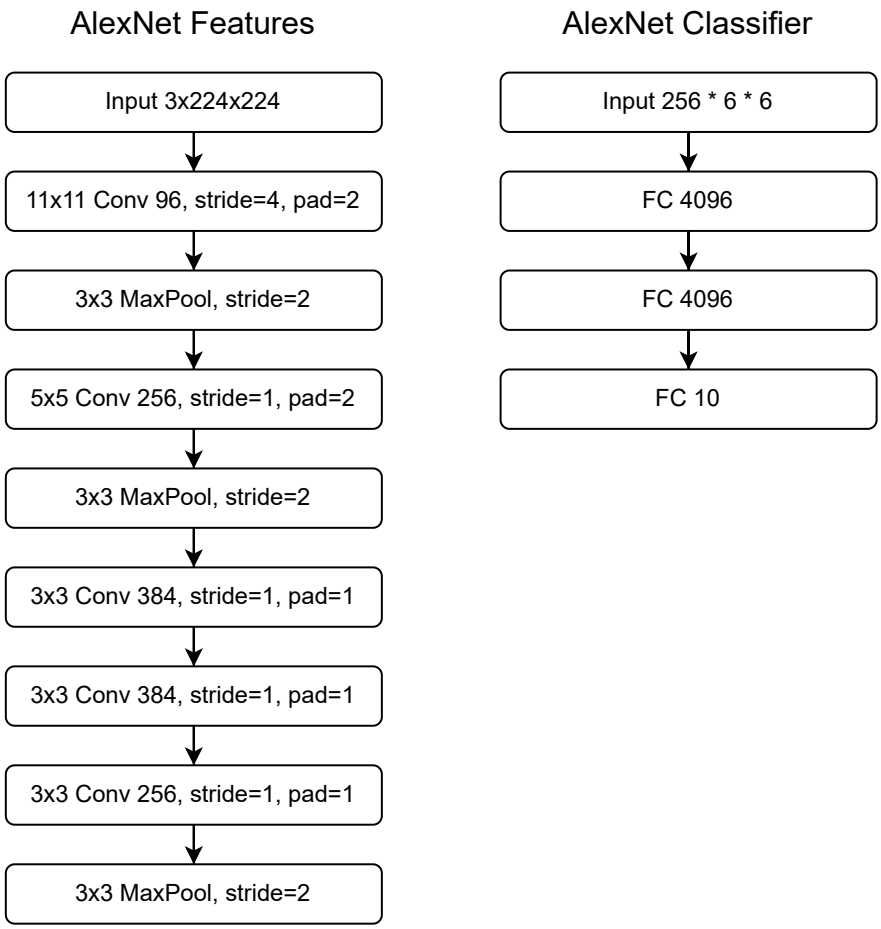


**Рисунок 9** – Схема архитектуры AlexNet

В оригинальной модели AlexNet после первых двух сверточных слоев используется ”local response” нормализация. Помимо классической модели был реализован вариант с использованием более общего алгоритма ”batch” нормализации. В качестве оптимизатора обеих моделей использовался стохастический градиентный спуск с коэффициентом момента $0.9$ и размером batch равным $128$.

Оригинальная модель показала лучший результат (на основе $10$ запусков обучения) по сравнению с модификацией, которая использует batch нормализацию: $0.843$ против $0.83$ соответственно. Результаты обучения моделей пред-

ставлены на рисунках 10 и 11.

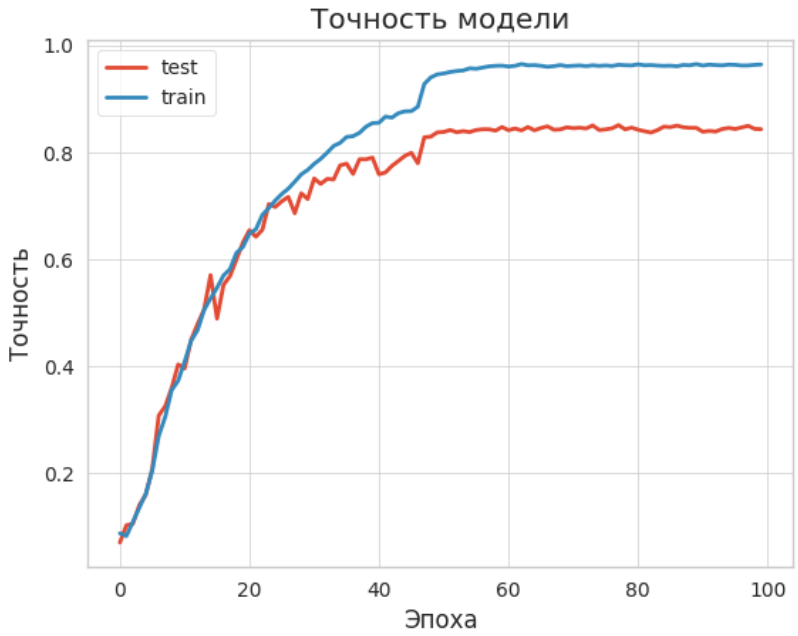


**Рисунок 10** – График точности модели AlexNet с local response нормализацей

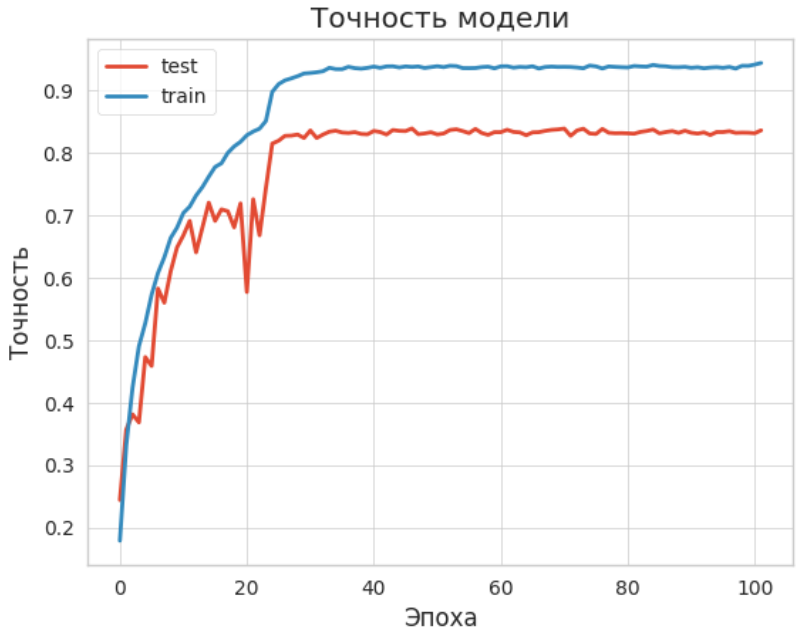


**Рисунок 11** – График точности модели AlexNet с batch нормализацей

Также к моделям был применен алгоритм визуализации Grad-CAM. На рисунках 12 и 13 представлен результат работы алгоритма для последнего свер-точного слоя варианта с local response нормализацией.

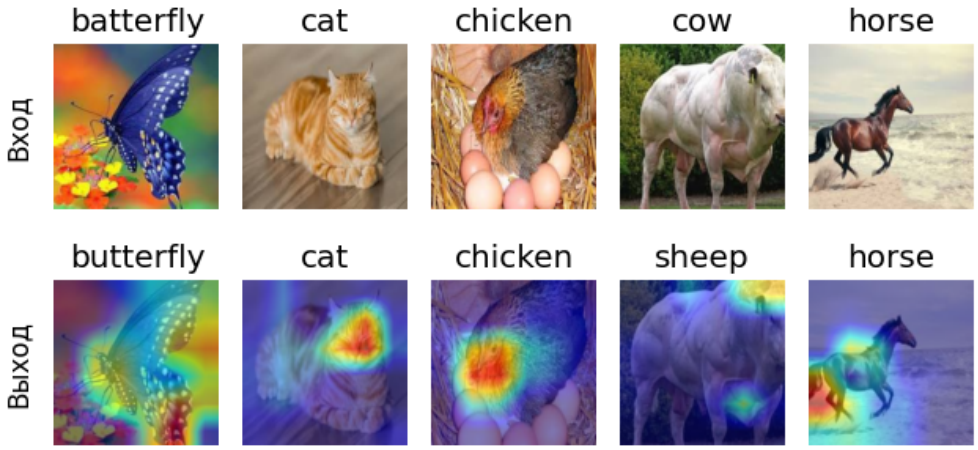


**Рисунок 12** – Тепловые карты Grad-CAM для модели AlexNet

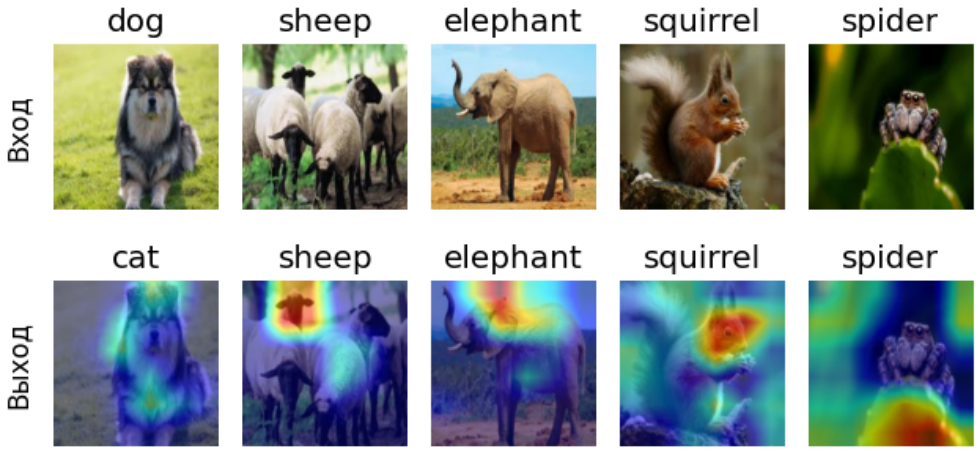


**Рисунок 13** – Тепловые карты Grad-CAM для модели AlexNet (продолжение)

Используя Grad-CAM можно сделать некоторые выводы о признаках, которые ”выучила” модель. Например, класс бабочки определяют ее крылья, класс лошади – ее задние ноги и хвост, а класс паука предсказывается не по самому пауку, а по его окружению. Последняя ситуация крайне вредна для модели, потому что схожий фон может быть у изображений и других животных. Без алгоритма визуализации такое поведение модели предсказать было бы очень сложно.

В листинге 5 приложения представлена програмная реализация архитектуры AlexNet.

## 2.3 Архитектура VGGNet

В отличие от AlexNet, архитектура VGG не использует большие ядра сверток. Вместо этого, используется больше слоев с малыми ядрами размера $3 \times 3$. Такой подход имеет ряд преимуществ [8]. Во первых, несколько слоев с малыми свертками имееют намного меньше параметров: на один канал сверточного слоя со сверткой $121 \times 121$ необходимо 121 параметров, в то время как для 3 сверточных слоев необходимо всего $9 + 9 + 9 = 27$ параметров. Во вторых, рецептивное поле 3 малого сверточного слоя имеет размер $3 + 2 + 2 = 7$ клеток, что соотвествует рецептивному полю сверточного слоя с ядром $7 \times 7$, имеющее большее число параметров.

Глубокие сети также выделяют более сложные признаки, качественно отличающиеся от признаков ”широких” сетей вроде AlexNet, что будет продемонстрировано позже.

VGG на самом деле являются целым семейством моделей, поскольку единообразная организация сверточных слоев позволяет строить похожие модели разной глубины и ширины. На рисунке 14 изображена модель VGG13 (10 сверточных + 3 полносвязных слоя).

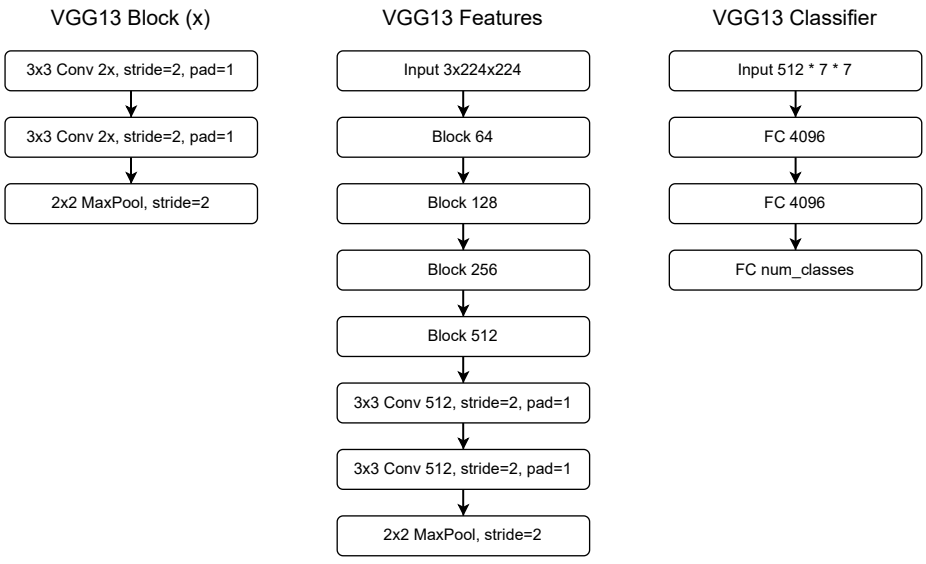


**Рисунок 14** – Модель VGG13

Для рассматриваемого датасета предложена модификация модели VGG13. В отличие от оригинала, модификация CVGG13 использует меньшее число параметров полносвязных слоев, а также использует batch нормализацию на каждом сверточном слое. Меньший размер полносвязных слоев обусловлен малым размером тренировочных данных, на которых крупная модель (оригинальная VGG13 имеет порядка 130 млн. параметров) не сможет корректно обучиться. Схема модифицированной модели представлена на рисунке 15.

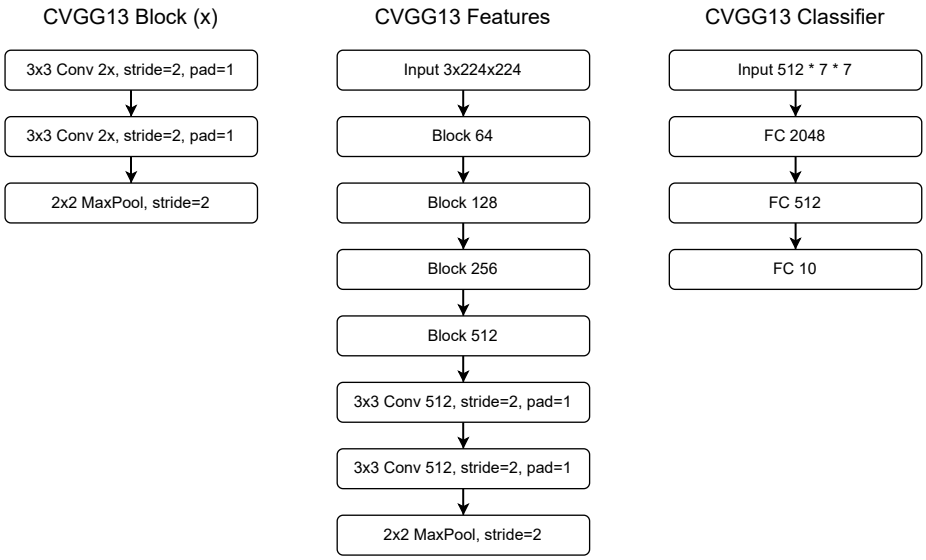


**Рисунок 15** – Модель CVGG13

Для обучения CVGG13 использовался стохастический градиентный спуск с коэффициентом момента 0.9 и batch размера 64. Модель получила отличные результаты как в точности, равной 0.915, так и в интерпретируемости признаков с помощью Grad-CAM. Результаты обучения представлены на рисунке 16.

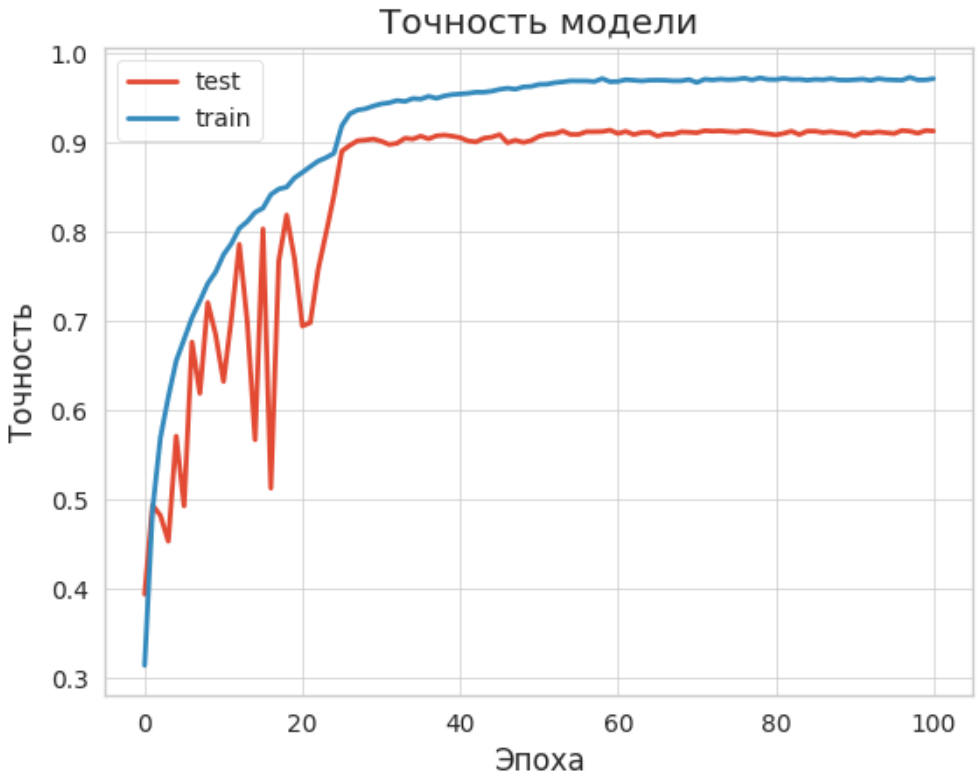


**Рисунок 16** – График изменения точности CVGG13

Результаты применения Grad-CAM к последнему слою разработанной модели представлены на рисунках 17 и 18.

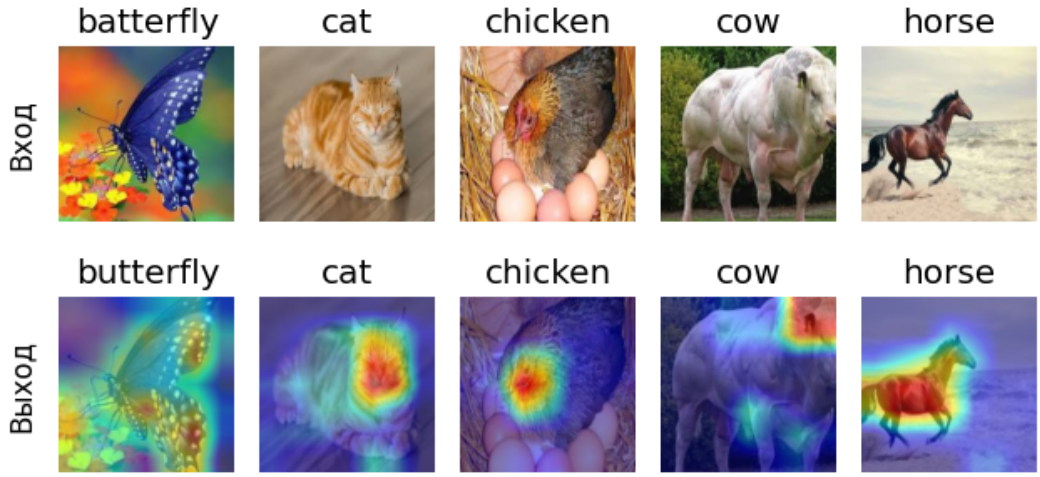


**Рисунок 17** – Тепловые карты Grad-CAM для модели CVGG13

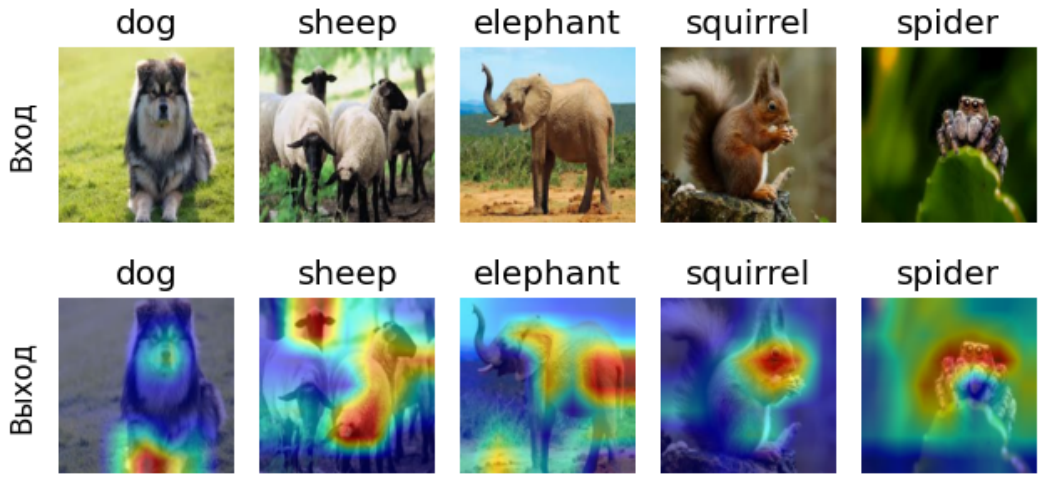


**Рисунок 18** – Тепловые карты Grad-CAM для модели CVGG13 (продолжение)

Изученные моделью признаки намного ”крепче”, чем признаки модели AlexNet. В отличие от последней, класс паука определяется по самому пауку, а не его окружению. Данная модель также показала более качественные тепловые карты и для остальных классов, особенно овцы и собаки.

Програмная реализация моделей семейства VGG13 представлена в листинге 4 приложения А.

# ЗАКЛЮЧЕНИЕ

В результате работы были изучены архитектуры моделей сверточных нейронных сетей AlexNet и VGG, общие методы глубокого обучения, метод интерпретирования сверточных моделей Grad-CAM. Были предложены модификации указанных архитектур, решающие поставленную в работе задачу по классификации изображений.

В рамках работы выяснились недостатки AlexNet по сравнению с VGG13. С помощью Grad-CAM выяснилось, что часть классов модель AlexNet определяет по окружению, а не по животному. CVGG13 при этом лишена таких недостатков и рекомендуется к использованию в задаче классификации представленых в датасете 10 классов.

Полученная моделями точность (>80%) является более чем удовлетворительной для малого обучающего набора. Таким образом, была достигнута цель работы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Goodfellow I., Bengio Y., Courville A. Deep Learning [Электронный ресурс]. — MIT Press, 2016. — URL: http://www.deeplearningbook.org.

2. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization [Электронный ресурс] / R.R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra // International Journal of Computer Vision. — 2019. — Vol. 128, № 2. — P. 336–359. — URL: http://dx.doi.org/10.1007/s11263-019-01228-7.

3. Воронцов К.В. Курс лекций по машинному обучению [Электронный ресурс]. — URL: http://www.machinelearning.ru/wiki/index.php?title=Машинное_обучение_\%28курс_лекций\%2C_К.В.Воронцов\%29

4. Rumelhart D.E., Hinton G.E., Williams R.J. Learning representations by back-propagating errors [Электронный ресурс] // Nature. — 1986. — Vol. 323, № 6088. — P. 533–536. — URL: https://doi.org/10.1038/323533a0

5. Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [Электронный ресурс]. — 2015. — URL: https://arxiv.org/abs/1502.03167

6. Srivastava N., Hinton G., Krizhevsky A., Sutskever I., Salakhutdinov R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting [Электронный ресурс] // Journal of Machine Learning Research. — 2014. — Vol. 15, № 56. — P. 1929–1958. — URL: http://jmlr.org/papers/v15/srivastava14a.html

7. Krizhevsky A., Sutskever I., Hinton G.E. ImageNet Classification with Deep Convolutional Neural Networks [Электронный ресурс] // Advances in Neural Information Processing Systems / Под ред. F. Pereira, C.J. Burges, L. Bottou, K.Q. Weinberger. — Curran Associates, Inc., 2012.

— Vol. 25. — URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

8. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition [Электронный ресурс]. — 2015. — URL: https://arxiv.org/abs/1409.1556

# ПРИЛОЖЕНИЕ А

**Листинг 1** – Файл utils/__init__.py

```python
VGG_MODELS = ["vgg13", "cvgg13", "vgg16"]
RESNET_MODELS = ["resnet18", "resnet18_pretrained", "resnet50"]
ALEXNET_MODELS = ["alexnet_lrn", "alexnet_bn"]


def get_model(model_name):
    from models.vgg import CVGG13, VGG13, VGG16
    from models.alexnet import AlexNetLRN, AlexNetBN
    from models.resnet import resnet18, resnet50

    match model_name:
        case "vgg13":
            return VGG13()
        case "vgg16":
            return VGG16()
        case "cvgg13":
            return CVGG13()
        case "alexnet_lrn":
            return AlexNetLRN()
        case "alexnet_bn":
            return AlexNetBN()
        case "resnet18":
            return resnet18()
        case "resnet18_pretrained":
            return resnet18(weights="DEFAULT")
        case "resnet50":
            return resnet50()


def get_last_layer(model, model_name):
    if model_name in VGG_MODELS or model_name in ALEXNET_MODELS:
        return model.features[-1]
    elif model_name in RESNET_MODELS:
        return model.layer4[-1]
    else:
        raise ValueError(f"Unknown last layer for {model_name}")


def get_transforms():
    from torchvision import transforms

    return transforms.Compose(
        [
            transforms.Resize((256, 256)),
            transforms.CenterCrop((224, 224)),
        ]
    )

```

```
49
50 def get_eval_transforms():
51     from torchvision import transforms
52
53     return transforms.Compose(
54         [
55             get_transforms(),
56             transforms.ToTensor(),
57         ]
58     )
```

**Листинг 2** – Файл utils/train.py

```python
import torch
from sklearn.metrics import f1_score
from rich.progress import track
from .checkpoint import create_checkpoint


@torch.compile
def train_epoch(model, criterion, loader, device, optimizer, scaler):
    model.train()
    loss_epoch = 0.0
    total_samples = 0
    total_true = []
    total_pred = []

    for images, labels in loader:
        # setup
        images = images.to(device, non_blocking=True)
        labels = labels.to(device, non_blocking=True)
        batch_size = images.size(0)

        # forward
        optimizer.zero_grad()
        with torch.autocast(device, dtype=torch.float16):
            outs = model(images)
            loss = criterion(outs, labels)

        # backward
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # stats
        with torch.no_grad():
            preds = torch.argmax(outs, dim=1)
            loss_epoch += loss.item() * batch_size
            total_samples += batch_size
            total_true.append(labels.cpu())
            total_pred.append(preds.cpu())

    total_true = torch.cat(total_true).numpy()
    total_pred = torch.cat(total_pred).numpy()

    loss_epoch /= total_samples
    f1_epoch = f1_score(total_true, total_pred, average="micro")
    return loss_epoch, f1_epoch


@torch.compile
@torch.no_grad()
def test_epoch(model, criterion, loader, device):
    model.eval()
```

```
    loss_epoch = 0.0
    total_samples = 0
    total_true = []
    total_pred = []

    for images, labels in loader:
        # setup test data
        images = images.to(device, non_blocking=True)
        labels = labels.to(device, non_blocking=True)
        batch_size = images.size(0)

        with torch.autocast(device, dtype=torch.float16):
            outs = model(images)
            loss = criterion(outs, labels)

        # stats
        preds = torch.argmax(outs, dim=1)
        loss_epoch += loss.item() * batch_size
        total_samples += batch_size
        total_true.append(labels.cpu())
        total_pred.append(preds.cpu())

    total_true = torch.cat(total_true).numpy()
    total_pred = torch.cat(total_pred).numpy()

    loss_epoch /= total_samples
    f1_epoch = f1_score(total_true, total_pred, average="micro")
    return loss_epoch, f1_epoch


def train(
    model,
    train_loader,
    test_loader,
    device,
    criterion,
    optimizer,
    scaler,
    scheduler=None,
    writer=None,
    epochs=10,
    start_epoch=0,
    checkpoint_dir="data/checkpoints",
    checkpoint_step=10,
):
    for epoch in track(
        range(start_epoch, start_epoch + epochs), description="Training..."
    ):
        # train
        loss, f1 = train_epoch(
            model=model,
            loader=train_loader,
            device=device,
```

```
            optimizer=optimizer,
            scaler=scaler,
            criterion=criterion,
        )

        # test
        test_loss, test_f1 = test_epoch(
            model=model,
            criterion=criterion,
            loader=test_loader,
            device=device,
        )

        if writer:
            writer.add_scalar("Loss/train", loss, epoch)
            writer.add_scalar("Loss/test", test_loss, epoch)
            writer.add_scalar("F1-score/train", f1, epoch)
            writer.add_scalar("F1-score/test", test_f1, epoch)

        if scheduler:
            if isinstance(scheduler, torch.optim.lr_scheduler.ReduceLROnPlateau):
                scheduler.step(test_loss)
            else:
                scheduler.step()

        # checkpoint
        if (epoch + 1) % checkpoint_step == 0:
            create_checkpoint(
                dir=checkpoint_dir,
                model=model,
                epoch=epoch + 1,
                optimizer=optimizer,
                scaler=scaler,
                scheduler=scheduler,
            )

    return start_epoch + epochs
```

**Листинг 3** – Файл utils/checkpoint.py

```python
import torch
import os
from pathlib import Path


def save_weights(model, model_name, dir="data/weights"):
    save_path = Path(dir) / f"{model_name}.pt"
    torch.save(model.state_dict(), save_path)


def create_checkpoint(
    dir,
    model,
    epoch,
    optimizer=None,
    scaler=None,
    scheduler=None,
):
    save_dict = {
        "model_state_dict": model.state_dict(),
        "epoch": epoch,
    }

    if optimizer:
        save_dict["optimizer_state_dict"] = optimizer.state_dict()

    if scaler:
        save_dict["scaler_state_dict"] = scaler.state_dict()

    if scheduler:
        save_dict["scheduler_state_dict"] = scheduler.state_dict()

    # checkpoint dir
    dir_path = Path.cwd() / dir
    try:
        dir_path.mkdir(parents=True)
    except:
        pass

    save_path = dir_path / f"checkpoint_{epoch}.pt"
    link_path = dir_path / "checkpoint.pt"
    torch.save(save_dict, save_path)

    # link last checkpoint
    try:
        os.remove(link_path)
    except:
        pass
    os.symlink(save_path, link_path)


```

```python
def load_checkpoint(
    dir,
    model,
    optimizer=None,
    scaler=None,
    scheduler=None,
    name="checkpoint.pt",
):
    checkpoint = torch.load(Path(dir) / name)

    model.load_state_dict(checkpoint["model_state_dict"])

    if optimizer and "optimizer_state_dict" in checkpoint:
        optimizer.load_state_dict(checkpoint["optimizer_state_dict"])

    if scaler and "scaler_state_dict" in checkpoint:
        scaler.load_state_dict(checkpoint["scaler_state_dict"])

    if scheduler and "scheduler_state_dict" in checkpoint:
        scheduler.load_state_dict(checkpoint["scheduler_state_dict"])

    return checkpoint["epoch"]
```

**Листинг 4** – Файл models/vgg.py

```
import torch.nn as nn


class VGGConvBlock(nn.Module):
    def __init__(
        self,
        in_channels,
        out_channels,
        pool,
    ):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3,
            padding="same")
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        if pool:
            self.maxpool = nn.MaxPool2d(kernel_size=2)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = x if not hasattr(self, "maxpool") else self.maxpool(x)
        return x


class CVGG13(nn.Module):
    def __init__(self, drop_rate=0.5):
        super().__init__()

        self.features = nn.Sequential(
            VGGConvBlock(3, 64, pool=False),
            VGGConvBlock(64, 64, pool=True),
            VGGConvBlock(64, 128, pool=False),
            VGGConvBlock(128, 128, pool=True),
            VGGConvBlock(128, 256, pool=False),
            VGGConvBlock(256, 256, pool=True),
            VGGConvBlock(256, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
            VGGConvBlock(512, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
        )

        self.flatten = nn.Flatten()

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 2048),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_rate),
            nn.Linear(2048, 512),
            nn.ReLU(inplace=True),
```

```
            nn.Dropout(drop_rate),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.flatten(x)
        x = self.classifier(x)
        return x


class VGG13(nn.Module):
    def __init__(self, drop_rate=0.5):
        super().__init__()

        self.features = nn.Sequential(
            VGGConvBlock(3, 64, pool=False),
            VGGConvBlock(64, 64, pool=True),
            VGGConvBlock(64, 128, pool=False),
            VGGConvBlock(128, 128, pool=True),
            VGGConvBlock(128, 256, pool=False),
            VGGConvBlock(256, 256, pool=True),
            VGGConvBlock(256, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
            VGGConvBlock(512, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
        )

        self.flatten = nn.Flatten()

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_rate),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_rate),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.flatten(x)
        x = self.classifier(x)
        return x


class VGG16(nn.Module):
    def __init__(self, drop_rate=0.5):
        super().__init__()

        self.features = nn.Sequential(
            VGGConvBlock(3, 64, pool=False),
```

```
            VGGConvBlock(64, 64, pool=True),
            VGGConvBlock(64, 128, pool=False),
            VGGConvBlock(128, 128, pool=True),
            VGGConvBlock(128, 256, pool=False),
            VGGConvBlock(256, 256, pool=False),
            VGGConvBlock(256, 256, pool=True),
            VGGConvBlock(256, 512, pool=False),
            VGGConvBlock(512, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
            VGGConvBlock(512, 512, pool=False),
            VGGConvBlock(512, 512, pool=False),
            VGGConvBlock(512, 512, pool=True),
        )

        self.flatten = nn.Flatten()

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_rate),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(drop_rate),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.flatten(x)
        x = self.classifier(x)
        return x
```

**Листинг 5** – Файл models/alexnet.py

```
import torch.nn as nn


class AlexNetBlockLRN(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super().__init__()

        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride,
            padding=2)
        self.relu = nn.ReLU(inplace=True)
        self.lrn = nn.LocalResponseNorm(size=5)
        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        x = self.lrn(x)
        x = self.pool(x)
        return x


class AlexNetBlockBN(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super().__init__()

        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, stride,
            padding=2)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.pool(x)
        return x


class AlexNetBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.relu(x)
        return x

```

```
50
51 class AlexNetLRN(nn.Module):
52     def __init__(self, dropout=0.5):
53         super().__init__()
54
55         self.features = nn.Sequential(
56             AlexNetBlockLRN(3, 96, 11, 4),
57             AlexNetBlockLRN(96, 256, 5, 1),
58             AlexNetBlock(256, 384),
59             AlexNetBlock(384, 384),
60             AlexNetBlock(384, 256),
61             nn.MaxPool2d(kernel_size=3, stride=2),
62         )
63
64         self.flatten = nn.Flatten()
65
66         self.classifier = nn.Sequential(
67             nn.Linear(256 * 6 * 6, 4096),
68             nn.ReLU(inplace=True),
69             nn.Dropout(dropout),
70             nn.Linear(4096, 4096),
71             nn.ReLU(inplace=True),
72             nn.Dropout(dropout),
73             nn.Linear(4096, 10),
74         )
75
76     def forward(self, x):
77         x = self.features(x)
78         x = self.flatten(x)
79         x = self.classifier(x)
80         return x
81
82
83 class AlexNetBN(nn.Module):
84     def __init__(self, dropout=0.5):
85         super().__init__()
86
87         self.features = nn.Sequential(
88             AlexNetBlockBN(3, 96, 11, 4),
89             AlexNetBlockBN(96, 256, 5, 1),
90             AlexNetBlock(256, 384),
91             AlexNetBlock(384, 384),
92             AlexNetBlock(384, 256),
93             nn.MaxPool2d(kernel_size=3, stride=2),
94         )
95
96         self.flatten = nn.Flatten()
97
98         self.classifier = nn.Sequential(
99             nn.Linear(256 * 6 * 6, 4096),
100             nn.ReLU(inplace=True),
101             nn.Dropout(dropout),
102             nn.Linear(4096, 4096),
```

```
            nn.ReLU(inplace=True),
            nn.Dropout(dropout),
            nn.Linear(4096, 10),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.flatten(x)
        x = self.classifier(x)
        return x
```

**Листинг 6** – Файл research.ipynb

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torch.amp import GradScaler
from torch.utils.tensorboard import SummaryWriter

from torchvision.datasets import ImageFolder
from torchvision import transforms

import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_class_weight

import logging
from utils.train import train
from utils.checkpoint import (
    load_checkpoint,
    create_checkpoint,
    save_weights,
)

from utils import get_model

torch._logging.set_logs(all=logging.ERROR)
torch.multiprocessing.set_start_method("spawn", force=True)

MODEL = "resnet50"
BATCH_SIZE = 128
EPOCHS = 100
SEED = 11
LR = 1e-2
WEIGHT_DECAY = 1e-4
LOAD = True
CHECKPOINT_NAME = "checkpoint.pt"

torch.manual_seed(SEED)

transform = transforms.Compose(
    [
        transforms.Resize((256, 256)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(20),
        transforms.RandomCrop((224, 224)),
        transforms.ToTensor(),
    ]
)

dataset = ImageFolder(
    "data/images/",
    transform=transform,
```

```
    )

    class_weight = compute_class_weight(
        "balanced",
        classes=np.arange(10),
        y=dataset.targets,
    )

    class_weight = torch.tensor(class_weight, dtype=torch.float32)

    train_indices, test_indices = train_test_split(
        range(len(dataset.targets)),
        test_size=0.2,
        stratify=dataset.targets,
        random_state=SEED,
    )

    train_dataset = Subset(dataset, train_indices)
    test_dataset = Subset(dataset, test_indices)

    train_loader = DataLoader(
        train_dataset,
        batch_size=BATCH_SIZE,
        shuffle=True,
        num_workers=4,
        persistent_workers=True,
        pin_memory=True,
    )

    test_loader = DataLoader(
        test_dataset,
        batch_size=BATCH_SIZE,
        shuffle=False,
        num_workers=4,
        persistent_workers=True,
        pin_memory=True,
    )

    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = get_model(MODEL).to(device)
    model.compile(dynamic=False, mode="max-autotune")
    criterion = nn.CrossEntropyLoss(weight=class_weight.to(device))
    optimizer = optim.AdamW(
        model.parameters(),
        lr=LR,
        weight_decay=WEIGHT_DECAY,
    )
    scheduler = optim.lr_scheduler.MultiStepLR(
        optimizer,
        milestones=[30, 60],
        gamma=0.1,
    )
    scaler = GradScaler(device)
```

```
    writer = SummaryWriter(log_dir=f"logs/{MODEL}")
    epoch = 0

    if LOAD:
        try:
            epoch += load_checkpoint(
                dir=f"data/checkpoints/{MODEL}",
                model=model,
                optimizer=optimizer,
                scaler=scaler,
                scheduler=scheduler,
                name=CHECKPOINT_NAME,
            )
        except:
            pass

    epoch = train(
        model=model,
        train_loader=train_loader,
        test_loader=test_loader,
        device=device,
        criterion=criterion,
        optimizer=optimizer,
        scaler=scaler,
        scheduler=scheduler,
        writer=writer,
        epochs=EPOCHS,
        start_epoch=epoch,
        checkpoint_dir=f"data/checkpoints/{MODEL}",
    )

    create_checkpoint(
        dir=f"data/checkpoints/{MODEL}",
        model=model,
        epoch=epoch,
        optimizer=optimizer,
        scaler=scaler,
        scheduler=scheduler,
    )

    save_weights(model, MODEL)
    writer.close()
```

**Листинг 7** – Файл visualize.ipynb

```
import torch
import matplotlib.pyplot as plt
from PIL import Image
from pathlib import Path
from utils import get_transforms, get_model, get_last_layer
from cnn import run_model, apply_gradcam

plt.style.use("ggplot")

MODEL = "alexnet_lrn"

images_dir = Path("data/gradcam_images/ref")
transform = get_transforms()
device = "cuda" if torch.cuda.is_available() else "cpu"
model = get_model(MODEL).to(device)
model.load_state_dict(torch.load(f"data/weights/{MODEL}.pt"))
layer = get_last_layer(model, MODEL)

inputs = []
outputs = []


for i, path in enumerate(images_dir.iterdir()):
    image = transform(Image.open(path))
    label = path.stem
    inputs.append((image, label))

    gradcam_image = apply_gradcam(model, device, image, layer)
    prediction = run_model(model, device, image)[0]
    outputs.append((gradcam_image, prediction))

def visualize(inputs, outputs):
    n = len(inputs)
    fig, axes = plt.subplots(2, n)

    for ax in axes.flat:
        ax.axis("off")

    for i in range(n):
        image, label = inputs[i]
        gradcam_image, prediction = outputs[i]

        axes.flat[i].imshow(image)
        axes.flat[i].set_title(label)
        axes.flat[i + n].imshow(gradcam_image)
        axes.flat[i + n].set_title(prediction)

    axes[0, 0].annotate(
        "Вход",
        xy=(-0.3, 0.5),
        xycoords="axes fraction",
```

```
            rotation=90,
            va="center",
            fontsize=12,
        )

        axes[1, 0].annotate(
            "Выход",
            xy=(-0.3, 0.5),
            xycoords="axes fraction",
            rotation=90,
            va="center",
            fontsize=12,
        )

        plt.tight_layout()
        plt.subplots_adjust(hspace=-0.5)
        plt.show()

    visualize(inputs[:5], outputs[:5])

    visualize(inputs[5:], outputs[5:])

    import polars as pl
    import seaborn as sns

    test = pl.read_csv(f"data/csv/f1_test_{MODEL}.csv")
    train = pl.read_csv(f"data/csv/f1_train_{MODEL}.csv")

    sns.lineplot(test, x="Step", y="Value", label="test")
    sns.lineplot(train, x="Step", y="Value", label="train")
    plt.title("F1-score")
    plt.show()
```

**Листинг 8** – Файл cnn.py

```
#!/usr/bin/env python

import torch
from PIL import Image
from argparse import ArgumentParser
from utils import (
    get_model,
    get_eval_transforms,
    get_last_layer,
    VGG_MODELS,
    RESNET_MODELS,
    ALEXNET_MODELS,
)
from pytorch_grad_cam import GradCAM
from pytorch_grad_cam.utils.image import show_cam_on_image


def logits_to_class(logits):
    idx_to_class = [
        "butterfly",
        "cat",
        "chicken",
        "cow",
        "dog",
        "elephant",
        "horse",
        "sheep",
        "spider",
        "squirrel",
    ]

    predicted_class = torch.argmax(logits).item()
    return idx_to_class[predicted_class]


@torch.no_grad()
def run_model(model, device, image):
    model.eval()
    transform = get_eval_transforms()
    x = transform(image).unsqueeze(0).to(device)
    logits = model(x)
    return logits_to_class(logits), logits


def apply_gradcam(model, device, image, layer):
    model.eval()
    transform = get_eval_transforms()
    input = transform(image).unsqueeze(0).to(device)
    target_layers = [layer]

    cam = GradCAM(model=model, target_layers=target_layers)
```

```
    targets = None
    grayscale_cam = cam(input, targets=targets, aug_smooth=True)[0, :]
    rgb_img = input.squeeze(0).permute((1, 2, 0)).cpu().numpy()
    visualization = show_cam_on_image(rgb_img, grayscale_cam, use_rgb=True)
    return Image.fromarray(visualization)


if __name__ == "__main__":
    parser = ArgumentParser(prog="model")

    parser.add_argument(
        "filename",
        help="File to classify",
    )

    parser.add_argument(
        "-d",
        "--device",
        default="cuda" if torch.cuda.is_available() else "cpu",
        choices=["cuda", "cpu"],
        help="Device: cpu or gpu(cuda)",
    )

    parser.add_argument(
        "-m",
        "--model",
        default="cvgg13",
        choices=VGG_MODELS + RESNET_MODELS + ALEXNET_MODELS,
        help="Model to use",
    )

    args = parser.parse_args()
    image = Image.open(args.filename)
    model = get_model(args.model).to(args.device)
    model.load_state_dict(torch.load(f"data/weights/{args.model}.pt"))

    prediction = run_model(model, args.device, image)[0]
    print(prediction)

    layer = get_last_layer(model, args.model)
    gradcam_image = apply_gradcam(model, args.device, image, layer)
    gradcam_image.show()
```