6. ЛАБОРАТОРНАЯ РАБОТА № 6 «ИССЛЕДОВАНИЕ НЕЙРОННЫХ СЕТЕЙ»

6.1. Цель работы

Исследование архитектур различных нейронных сетей, функций потерь, методов машинного обучения, основанных на градиентном спуске, приобретение навыков разработки, обучения и программирования архитектур нейронных сетей для решения задач нелинейной регрессии, классификации рукописных цифр и идентификации языка

6.2. Краткие теоретические сведения

6.2.1. Машинное обучение

В машинном обучении системы обучаются, а не программируются явно. В качестве обучаемых систем широко используются модели искусственных нейронных сетей (ИНС), структурированные в виде слоёв. Им передаются многочисленные примеры данных, имеющие отношение к решаемой задаче, а ИНС находят в этих примерах статистическую структуру, которая позволяет вырабатывать правила для автоматического решения задачи. Существуют три вида алгоритмов машинного обучения — это **алгоритмы обучения с учителем** (supervised learning algorithms), **алгоритмы обучения без учителя** (unsupervised learning algorithms), **алгоритмы обучения с подкреплением** (reinforcement learning) [10].

При обучении с учителем предполагается, что для каждого входного вектора ИНС мы заранее знаем желаемый выходной вектор и используем «отличие» действительного выходного вектора от желаемого для изменения в нужном направлении **параметров ИНС** (весов, смещений) , чтобы минимизировать «отличие». При обучении без учителя нам не известна желаемая реакция ИНС на заданный входной вектор. В этом случае ИНС должна «самообучаться», обнаруживая закономерности, присущие пространству входных данных. В этой лабораторной работе рассматриваются алгоритмы обучения с учителем.

В ходе обучения используют **набор данных** (датасет), который разбивают на три подмножества: **обучающее** (training), **валидационное** (validation) и **тестовое** (test) подмножества. Обучающее помножество данных используется для фактического создания модели ИНС, отображающей входные данные в выходные. Валижационное подмножество (данные разработки) используется для оценки эффективности модели ИНС путем вычисления точности предсказаний с помощью ИНС. Если модель ИНС работает не так хорошо, как бы хотелось (для этого вычисляется функция потерь), то можно вернуться и обучить ее снова, скорректировав либо специальные параметры модели, называемые **гиперпараметрами** (скорость обучения, размер мини-батча, коэффициент регуляризации и др.), либо использовать другой алгоритм обучения, пока не будут получены удовлетворительные оценки эффективности ИНС. На заключительном этапе используйте обученную модель ИНС для

предсказания значений тестового подмножества данных и оценки итоговой точности модели. Тестовое подмножество — это часть данных, которая не используется в ходе обучения ИНС.

6.2.2. Линейные классификаторы. Бинарный персептрон

Основная идея линейного классификатора заключается в том, чтобы выполнить классификацию, используя линейную комбинацию признаков $f_i(\mathbf{x})$ классифицируемых данных **x**. В векторной форме мы можем записать это как скалярное произведение вектора весов **w** на вектор признаков данных **f**(**x**):

$$net_w(\mathbf{x}) = \sum_i w_i f_i(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}), \quad (6.1)$$

где $net_w(\mathbf{x})$ – сетевое (предактивационное) значение. Функция формирования вектора признаков **f** соответствует некоторой предварительной обработке данных с целью получения признаковых представлений данных.

Рассмотрим реализацию простого линейного классификатора – бинарного персептрона. **Бинарный персептрон** находит границу решения, которая разделяет обучающие данные на два класса. В этом случае, когда сетевое значение (6.1) для точки данных **x** положительное, персетрон классифицирует эту точку данных как точку с положительной меткой +1, а если значение отрицательное – то как точку с отрицательной меткой -1. В ходе обучения персептрона мы ищем наилучшие возможные веса **w** так, чтобы любая обучающая точка могла быть идеально классифицирована.

Алгоритм обучения бинарного персептрона:

1. Инициализируйте все веса нулевыми значениями: **w** = **0** ;

2. Для каждого обучающей точки данных **x** с признаками **f**(**x**) и истинной меткой класса $y* \in \{-1, +1\}$ повторяйте:

(a) Классифицируйте точку данных **x**, используя текущий вектор весов **w**:

$$y = \begin{cases} +1, если\ net_w(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) \geq 0 \\ -1, если\ net_w(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) < 0 \end{cases}, \quad (6.2)$$

где y предсказанная метка;

(b) Сравните предсказанную метку y с истинной меткой y^*:

- если $y = y^*$, ничего не делайте;
- иначе, если $y \neq y^*$, обновите веса: $\mathbf{w} \leftarrow \mathbf{w} + y*\mathbf{f}(\mathbf{x})$.

3. Если для каждой обучающий точки данных все метки предсказаны правильно, то завершите работу. В противном случае повторите шаг 2.

6.2.3. Линейная регрессия

Задачи регрессии — это задача машинного обучения, в которой выход модели y является непрерывной переменной. Признаки могут быть как непрерывными, так и категориальными. Обозначим набор входных признаков модели как **x**

$\in$ R^n т. е. $\mathbf{x} = (x_1, \ldots, x_n)$. Тогда модель прогнозирования, соответствующая линейной регрессии, запишется в виде:

$$net_w(\mathbf{x}) = w_0 + w_1 x_1 + \ldots + w_n x_n, \quad (6.3)$$

где w_i – веса модели, которые необходимо оценить. Вес w_0 называют свобоным членом (в нейронных сетях – смещением) модели. Иногда к вектору признаков $\mathbf{x}$ добавляют признак, значение которого равно 1, чтобы можно было записать модель (6.3) в виде скалярного произведения $\mathbf{w}^T\mathbf{x}$, где $\mathbf{x} \in R^{n+1}$. Для оценки качества обучения модели используется функции потерь L2, которая соответствует среднему квадрату ошибки предсказания [10]

$$Loss(\mathbf{w}) = \frac{1}{2N}\sum_{j=1}^{N}(y^j - net_w(\mathbf{x}^j))^2 = \frac{1}{2N}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2, \quad (6.4)$$

где каждая строка матрицы $\mathbf{X}$ – это вектор $\mathbf{x}^j$, соответствующий j-ой точке данных. Дифференцируя функцию потерь по $\mathbf{w}$ и приравнивая производные нулю, можно найти оценку оптимального вектора весов.

6.2.4. Алгоритмы оптимизации параметров

В общем случае для заданной функции потерь (6.4) может не существовать решения в замкнутой форме. В таких случаях используют градиентные методы для поиска оптимальных весов. Идея заключается в том, что градиент функции указывает направление самого крутого увеличения функции, а направление, обратное градиенту, указывает направление самого крутого спуска. Соответственно, для поиска минимума функции потерь может использоваться **алгоритм градиентного спуска** [10]:

1. Инициализировать веса **w** случайными значениями;
2. **While w** *не сошлось* **do**:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\frac{\partial Loss(\mathbf{w})}{\partial \mathbf{w}}.$$

Здесь α – скорость обучения, $\frac{\partial Loss(\mathbf{w})}{\partial \mathbf{w}}$ – градиент функции потерь. В соответствии с этим алгоритмом обновление весов на каждой итерации выполняют на вектор пропорциональный градиенту функции потерь.

Так как датасет обычно содержит большое количество точек данных, то вычисление градиента на каждой итерации для всех точек данных слишком затратно. Поэтому были предложены такие подходы, как стохастический и мини-блочный градиентный спуск. При **стохастическом градиентном спуске** на каждой итерации алгоритма используется только одна точка данных для вычисления градиента. Эта точка данных каждый раз случайным образом выбирается из набора данных. Учитывая, что мы используем только одну точку данных для оценки градиента, стохастический градиентный спуск может привести к зашумленным градиентам и, таким образом, затруднить сходимость. **Мини-блочный градиентный спуск** является компромиссом между стохастическим и обычным алгоритмом градиентного

спуска, поскольку он использует **мини-блок** (batch) размером *m* точек данных каждый раз для вычисления градиентов. Размер мини-блока *m* является гиперпараметром, выбираемым пользователем.

6.2.5. Логистическая регрессия

Логистическая регрессия позволяет предсказать категориальную переменную. Для этого линейная комбинация входных признаков преобразуется в вероятность с помощью логистической функции:

$$P(y \mid \mathbf{x}; \mathbf{w}) = 1 \Big/ (1 + e^{-\mathbf{w}^T \mathbf{x}}). \quad (6.5)$$

Важно отметить что, хотя логистическая регрессия и называется регрессией, она используется для решения задач классификации, а не задач регрессии. Обратите внимание, что значения функции (6.5) находятся в интервале от 0 до 1. Интуитивно, логистическая функция моделирует вероятность принадлежности точки данных **x** к классу с меткой 1. Например, после того, как мы обучили логистическую регрессию, мы можем вычислить выход логистической функции для новой точки данных. Если значение выхода больше 0,5, мы классифицируем **x** как класс с меткой 1, а в противном случае — с меткой 0.

Оценить веса логистической регрессии можно, используя алгоритм градиентного спуска. При этом в качестве функции потерь используют отрицательную логарифмическую вероятность: – log(*P*(*y*| **x**; **w**)). Интуиция подсказывает, что хорошей (с низкими потерями) является модель, которая назначает высокую вероятность истинному выходу *y*, соответствующему входу **x** [10].

Многоклассовая логистическая регрессия позволяет классифицировать точки данных по *K* различным категориям, а не только по двум. В этом случае мы строим такую модель, которая даёт оценки вероятностей принадлежности точки данных к одной из *K* возможных категорий. Для этого вместо логистической функции используется **softmax** функция, которая определяет вероятность отнесения точки данных с признаками $\mathbf{f}(\mathbf{x}_i)$, к классу *j* как:

$$P(y_i = j \mid \mathbf{f}(\mathbf{x}_i); \mathbf{W}) = \frac{e^{\mathbf{w}_j{}^T \mathbf{f}(\mathbf{x}_i)}}{\sum_{k=1}^{K} e^{\mathbf{w}_k{}^T \mathbf{f}(\mathbf{x}_i)}}, \quad (6.6)$$

где **W** – матрица весов, каждый столбец которой содержит вектор весов $\boldsymbol{w}_k$. Аргументы экспоненты, называемые в данном случае **логитами** (logits), могут быть вычислены с помощью однослойной нейронной сети, содержащей *K* нейронов с весами $\boldsymbol{w}_k$ и вектором входа $\mathbf{f}(\mathbf{x}_i)$. В ходе обучения такой сети необходимо найти такую матрицу **W**, которая минимизирует функцию потерь в виде отрицательного логарифма вероятности

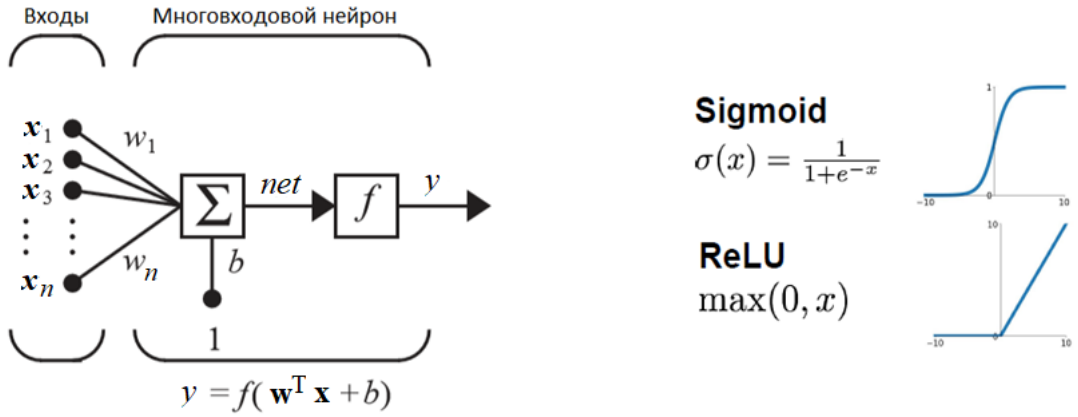
$$Loss(y_i; \mathbf{f}(\mathbf{x}_i)\mathbf{W}) = -\log P(y_i = i \mid \mathbf{f}(\mathbf{x}_i); \mathbf{W}). \quad (6.7)$$

Можно показать, что функция потерь (6.7) соответствует **кросс-энтропии**. Для случая 2-х классов выражение (6.7) соответствует **бинарной кросс-энтропии [**10**]:**

$$H(y_i; p_i) = -[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]. \quad (6.8)$$

6.2.6. Архитектура нейросетей

Элементарной ячейкой нейронной сети является нейрон. Отдельный нейрон с вектором входа **x**, состоящим из n элементов x_1, x_2, …, x_n, изображен на рисунке 6.1а.



а) *б*)

Рисунок 6.1 – Структура нейрона и функции активации

Здесь каждый элемент вектора данных **x** умножается на соответсвующий вес нейрона w_1, w_2, … , w_n Значения элементов вектора **x** , взвешенные с весами **w,** поступают на сумматор. Их сумма равна скалярному произведению вектора $\mathbf{w}^T$ на вектор входа **x**. К этой сумме добавляется смешение b. Результирующая сумма, называемая *сетевым (предактивационным) значением*, равна

$$net = w_1 x_1 + w_2 x_2 + \ldots + w_{1n} x_n + b = \mathbf{w}^T \mathbf{x} + b\,. \quad (6.9)$$

Сетевое значение *net* поступает на вход функции активации f, которая формирует выход нейрона

$$y = f(\mathbf{w}^T \cdot \mathbf{x} + b). \quad (6.10)$$

В качестве функций активации f в нейронных сетях часто используют **логистическую (сигмовидную)** и **линейную выпрямительную (ReLU)** функции, графики которых изображены на рисунке 6.1б.

При изображении нейросетей часто используется обобщенная векторно-матричная схема (рисунок 6.2). Вход нейрона изображается в виде темного прямоугольника, под которым указывается количество элементов n входного вектора **x**. Размер вектора входа **x** указывается ниже символа **x** и равен $n \times 1$. Вектор входа умножается на вектор $\mathbf{w}^T$ длины n. Константа 1 рассматривается как вход, который умножается на скалярное смещение b. Входом функции активации нейрона служит сумма смещения b и произведения $\mathbf{w}^T$ **x**. Эта сумма преобразуется функцией активации f в выходное значение нейрона y, которое в данном случае является скалярной величиной.

Структурная схема, изображенная на рисунке 6.2, соответствует однослойной сети с одним нейроном в слое. Если слой содержит *h* нейронов, то он характеризуется *матрицей весов* связей **W** размером *h* x *n*, вектором смещений **b** и вектором выхода **y** размером *h* x1. В этом случае каждый вход слоя связан с каждым нейроном слоя. Поэтому такой слой называют **полносвязным** (FC - fully-connected).

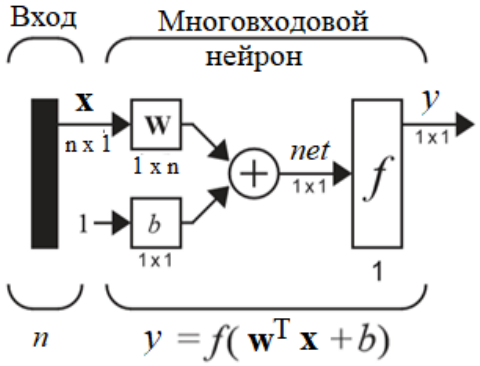


Рисунок 6.2 – Обобщенная векторно-матричная схема нейрона

С помощью схемы однослойной сети можно построить многослойную сеть. В качестве примера на рисунке 6.3 изображена трехслойная сеть прямого распространения (FF – feed forward) с с полносвязными слоями. Для этой сети выход предыдущего слоя является входом следующего слоя. Входом сети является вход первого слоя, т.е. вектор **x**, а выходом – выход последнего слоя, т.е. вектор $\mathbf{y}_3$. Соответственно, прямое распространение входного вектора по сети опишется выражением:

$$\mathbf{y}_3=\mathbf{f}_3(\mathbf{W}_3\mathbf{f}_2(\mathbf{W}_2\mathbf{f}_1(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)+\mathbf{b}_2)+\mathbf{b}_3. \qquad (6.11)$$

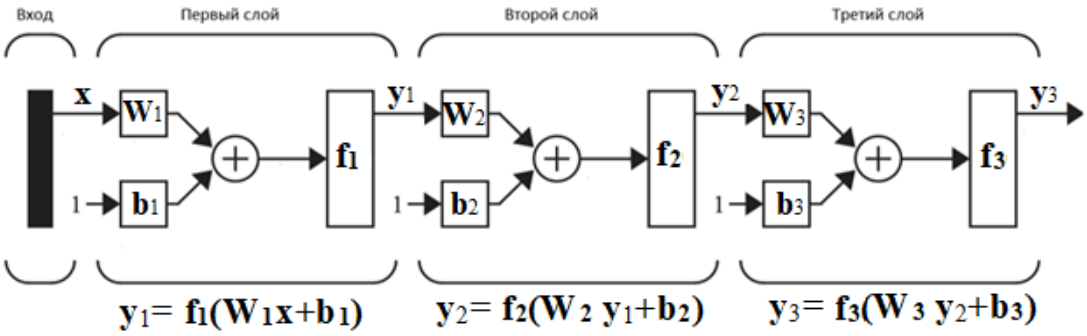


Рисунок 6.3 – Трехслойная сеть прямого распространения

Внутренние слои сети называются **скрытыми слоями** (hidden layers). Вход сети, представленный вектором **x**, называют *входным слоем*. Сокращенно структуру многослойной сети обозначают, указывая последовательно размерность входного слоя и количество нейронов в последующих слоях. Например, структура сети, изображенная на рисунке 6.3, обозначается в виде: n-h_1-h_2-h_3.

Простая нейронная сеть имеет линейные слои, где каждый линейный слой выполняет линейную операцию типа **Wx+b** или $\mathbf{x}^T\mathbf{W}+\mathbf{b}$. Линейные слои разделяются нелинейностями, что позволяет аппроксимировать с помощью нейросети

сложные функции **f(x)**, обеспечивающие автоматическое формирование признаковых представлений входных данных в ходе обучения нейросети. В лабораторной работе будем использовать в качестве нелинейности функцию ReLU, определяемую как relu(x)=max(x,0). Например, простая нейронная сеть с одним скрытым слоем с ReLU нелинейностью и выходным линейным слоем для отображения входного вектора-строки $\mathbf{x}^T$ в выходной вектор признаков $\mathbf{f}(\mathbf{x})$ может быть задана выражением:

$$\mathbf{f}(\mathbf{x})=\text{relu}(\mathbf{x}^T\cdot\mathbf{W}_1+\mathbf{b}_1)\cdot\mathbf{W}_2+\mathbf{b}_2\,, \qquad (6.11)$$

где $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{b}_1$, $\mathbf{b}_2$ – обучаемые параметры нейросети. В этом случае $\mathbf{W}_1$ будет матрицей размера n x h, где n — число элементов входного векторов **x**, а h — размер скрытого слоя (число нейронов скрытого слоя), $\mathbf{b}_1$ — вектор размера h. Мы вольны выбирать любое число нейронов скрытого слоя h (нужно только убедиться, что размеры других матриц и векторов согласованы, чтобы могли выполняться векторно-матричные операции). Использование больших значений h обычно делает сеть более ёмкой (способной вместить больше обучающих данных), но это может затруднить обучение сети (поскольку увеличивает число параметров, которые необходимо обучать), или может привести к переобучению на обучающих данных.

Для повышения вычислительной эффективности градиентного спуска обычно выполняют обработку не одного вектора **x,** а сразу целого блока входных данных. Это означает, что вместо одного входного вектора-строки $\mathbf{x}^T$ размером n на вход нейросети будет подаваться мини-блок (batch) входных данных размером m, представленный в виде матрицы **X** размером $m\times n$.

6.2.6. Фреймворк Pytorch

PyTorch — фреймворк/библиотека для разработки проектов глубокого обучения с упором на гибкость и возможность представлять модели глубокого обучения в характерном для Python стиле [11].

Ниже приведены основные функции, которые вам следует использовать при выполнении лабораторной работы. Этот список не является исчерпывающим. Все функции, которые вы можете использовать, импортированы для вас в модуле **models.py**. Для более детального знакомства с функциями обратитесь к документации PyTorch.

tensor(): Тензоры являются основной структурой данных в Pytorch. Они похожи на многомерные массивы Numpy, в том смысле, что вы можете складывать и умножать их. Каждый раз, когда вы используете функцию Pytorch, вы должны убедиться, что входные данные имеют форму тензора. Вы можете преобразовать список Python в тензор следующим образом: **tensor(data),** где **data** —n-элементный список. Также можно создать тензор на основе массива NumPy; **torch.from_numpy(npy),** где **npy** – массив NumPy.

relu(input): Активационная функция **relu**. Вызывается следующим образом: **relu(input)**. Она принимает входные данные **input** и возвращает **max(input, 0)**.

Linear: Класс для реализации линейного полносвязного нейросетевого слоя. Линейный слой обеспечивает для каждого нейрона слоя вычисление скалярного

произведение вектора весов нейрона и вектора входных данных. Вы должны инициализировать слой в конструктуре нейросети **__init__** следующим образом: **self. lin_layer = Linear(D_in, D_out)**, где **D_in** – число входов слоя, **D_out** – число выходов (нейронов) слоя. Вызов слоя выполняется при запуске модели нейросети с помощью инсрукции: **self. Linear_Layer (input)**. Для линейного слоя, определенного таким образом, Pytorch автоматически создает параметры слоя и обновляет их во время обучения.

movedim(тензор, начальные_позиции_измерений, конечные_позиции_измерений): Функция принимает **тензор** и меняет (переставляет местами) **начальные_позиции_измерений** (переданные как int) на **конечные_позиции_измерений**. Это будет полезно при выполнении задания 3.

cross_entropy(prediction, target): Функция потерь в виде кросс-энтропии (6.7) для многоклассовой классификации (задания 3–5). Здесь **prediction** — предсказанная метка класса моделью нейросети, **target** —истинная метка класса. Чем хуже предсказание нейросети, тем большее значение возвращает функция.

mse_loss(prediction, target): Среднеквадратическая функция потерь (6.4) для задачи регрессии (задание 2). Она используется аналогично функции **cross_entropy**.

Множества данных, используемые в лабораторной работе, доступны в виде объекта **dataset** Pytorch, который необходимо преобразовать в объект **dataloader**, чтобы с его помошью обеспечивать выборку мини-блоков из датасета:

```
data = DataLoader(training_dataset, batch_size = 64)
for batch in data:
   #Здесь размещается код обучения с использованием данных из batch
```

Каждый мини-блок, возвращаемый **DataLoader**, будет представлять собой словарь в форме: **{‘x’:features, ‘label’:label}**, где **label** — это истинная метка (и), которая должна предсказываться нейросетью на основе входных признаков **features**.

6.2.7. Разработка и программирование модели нейросети на PyTorch

Разработка нейронных сетей выполняется методом проб и ошибок. Ниже несколько советов, которые помогут вам при выборе архитектуры нейросети.

1. Будьте систематичны. Ведите журнал каждой архитектуры, которую вы опробовали, какими были гиперпараметры (размеры слоев, скорость обучения и т. д.) и какая эффективность сети была получена. По мере того, как вы набираетесь опыта, вы начнете видеть закономерности того, какие параметры имеют значение.
2. Начните с неглубокой сети (всего один скрытый слой, т. е. одна нелинейность). Более глубокие сети имеют экспоненциально больше комбинаций гиперпараметров, и даже одна ошибка может разрушить процесс обучения. Используйте небольшую сеть, чтобы найти подходящую скорость обучения и размер слоя; после этого вы можете рассмотреть возможность добавления большего количества слоев аналогичного размера.

3. Если скорость обучения выбрана неправильно, то ни один из других вариантов гиперпараметров не будет иметь значения. Слишком низкая скорость обучения приведет к медленному обучению модели, а слишком высокая может привести к тому, что процесс обучения будет расходиться. Начните с проверки разных скоростей обучения, наблюдая, как со временем уменьшаются потери. Для мини-блоков меньщих размеров требуются более низкие скорости обучения. Экспериментируя с размерами мини-блоков, имейте в виду, что наилучшая скорость обучения может для мини-блоков разных размеров отличаться.
4. Воздержитесь от создания слишком широкой сети (слишком большое число нейронов в скрытых слоях). Если вы будете расширять слои, точность будет постепенно снижаться, а время вычислений будет увеличиваться в квадрате. Полный автооценивание для всех частей задания занимает ~12 минут; если ваш код требует намного больше времени, вам следует проверить его на эффективность.
5. Если ваша модель возвращает **Infinity** или **NaN**, то, вероятно, скорость обучения слишком велика.

Рекомендуемые значения для гиперпараметров:

- размеры скрытых слоев: от 100 до 500;
- размер мини-блока: от 1 до 128, для заданий 2 и 3 необходимо, чтобы общий размер набора данных делился нацело на размер мини-блока;
- скорость обучения: от 0,0001 до 0,01;
- количество скрытых слоев: от 1 до 3.

В качестве простейшего примера программирования нейронной сети с помощью PyTorch рассмотрим определение коэффициентов линейной регрессии по набору точек данных. Начнем с четырех точек обучающих данных, полученных с использованием линейной функции $y=7x_0+8x_1+3$:

$$\mathbf{X}=\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 1 & 1\end{bmatrix},\quad \mathbf{y}=\begin{bmatrix}3\\ 11\\ 10\\ 18\end{bmatrix}.$$

Предположим, что данные предоставлены нам в виде тензоров.

```
>>> x
torch.Tensor([[0,0],[0,1],[1,0],[1,1])
>>> y
torch.Tensor([[3],[11],[10],[18]])
```

Построим и обучим модель вида $f(\mathbf{x})=x_0\cdot w_0+x_1\cdot w_1+b$. Если всё будет сделано правильно, мы должны будем получить, что $w_0=7$, $w_1=8$ и $b=3$.

Сначала создаем массивы обучаемых параметров модели нейросети. В матричной форме они могут быть записаны следующим образом:

$$\mathbf{W}=[\,w_0\ w_1],\quad \mathbf{b}=[b].$$

Для их хранения создаем тензоры в PyTorch:

```
w = Tensor(2, 1)
b = Tensor(1, 1)
```

Тензоры инициализируются нулевыми значениями. Поэтому если мы выведем значения **w** и **b**, то получим:

```
>>> w
torch.Tensor([[0],[0]])
>>> b
torch.Tensor([[0]])
```

Далее вычисляем предсказания *y* для нашей модели. Для этого сначала определяем линейный слой в конструкторе нейросети **__init__()**, как было указано выше с использованием класса **Linear**. Затем вычисляем предсказание

```
predicted_y = self.lin_layer(x)
```

Наша цель — добиться соответствия предсказанных значений **predicted_y** предоставленным данным **y.** В линейной регрессии мы делаем это путем минимизации средненго квадрата потерь, определяемых выражением (6.4). На PyTorch вычисление функции потерь (6.4) запишется следующим образом:

```
loss = mse_loss(predicted_y, y)
```

Для обучения нейронной сети нужно сначала инициализировать оптимизатор. В Pytorch встроено несколько оптимизаторов. Для этой лабораторной работы используйте **алгоритм оптимизации Adam** (вариант градиентного спуска с адаптацией скорости обучения):

```
optim.Adam(self.parameters(), lr=lr),
```

где **lr** — скорость обучения, **self.parameters()** — параметры нейросети, определяемые PyTorch автоматически, после создания экземпляра класса нейросети с помощью его конструктора.

Далее в цикле для каждой итерации по данным необходимо повторять следующие действия:

- обнулить градиенты, рассчитанные PyTorch с помощью вызова **optimizer.zero_grad()** ;
- вычислить тензор потерь **loss**, вызвав функцию **get_loss(),** которая вычисляет предсказание **predicted_y** и вызывает **mse_loss;**
- вычислить градиенты, используя вызов **loss.backward()**;
- обновить парамеры нейросети, вызвав **optimizer.step()**.

6.3. Задания для выполнения

Задание 1. Персептрон

В этом задании необходимо реализовать бинарный персептрон. Ваша задача — завершить реализацию класса **PerceptronModel** в **models.py**.

Для персептрона выходные метки должны иметь значение либо +1, либо −1.

Задание 2. Нелинейная регрессия

В этом задании необходимо разработать и обучить нейронную сеть аппроксимации функцию sin(x) на интервале $[-2\pi,2\pi]$. Вам нужно будет завершить реализацию класса **RegressionModel** в **models.py**.

Для этой задачи достаточно относительно простой архитектуры нейросети (см. рекоменлации в п. 6.2.7). В качестве функции потерь используйте функцию **mse_loss**.

Задание 3. Классификация цифр

В этом задании необходимо обучить сеть классификации рукописных цифр из набора данных MNIST. Каждая цифра представляется изображением размером 28 на 28 пикселей, значения которых хранятся в 784-мерном векторе в виде чисел с плавающей точкой. Выход нейросети представляет собой 10-мерный вектор, который имеет нули во всех позициях, за исключением единицы в позиции, соответствующей правильному классу цифры.

Завершите реализацию класса **DigitClassificationModel** в файле **models.py**. Значение, возвращаемое методом **DigitClassificationModel.run()** должно быть тензором размером **batch_size x 10**, содержащим оценки принадлежности цифры к определенному классу (0-9), где более высокие оценки указывают на более высокую вероятность принадлежности. В качестве функции потерь Вам следует использовать **cross_entropy**. Не включайте ReLU в последний слой сети, который должен быть линейным слоем.

Задание 4. Определение языка

Определение языка — это задача определения языка, на котором написан некоторый текст, по фрагменту текста. Например, ваш браузер может определить язык посещенной веб-страницы и предложить её перевести.

В этом задании необходимо построить модель нейронной сети, которая определяет язык очередного слова, поступающего на вход. Набор данных состоит из слов на пяти языках, как указано в таблице ниже:

Слово	**Язык**
discussed	Английский
eternidad	Испанский
itseänne	Финский
paleis	Голландский
mieszkać	Польский

Разные слова состоят из разного количества букв, поэтому модель нейросети должна иметь архитектуру, которая может обрабатывать слова переменной длины.

В этом задании на вход нейросети будет подаваться отдельные символы слова: x_0, x_1,⋯ , x_{L-1}, где L — длина слова.

Задание 5. Сверточная сеть

В этом задании создайте новую нейросеть для распознавания рукописных цифр в виде класса **DigitConvolutionalModel()** в **models.py**, сначала реализовав функцию **Convolve**. Эта функция принимает входную матрицу данных и матрицу весов и вычисляет их свертку.

6.4. Порядок выполнения лабораторной работы

6.4.1. Изучите по лекционному материалу и учебным пособиям [1-3, 9] основные понятия вероятностного вывода, понятие сетей Байеса, марковских моделей, методы и алгоритмы точного и приближенного вероятностного вывода в скрытых марковских моделях. Ответьте на контрольные вопросы.

6.4.2. Используйте для выполнения лабораторной работы файлы из архива **МиСИИ_лаб6_2024.zip.** Разверните программный код лабораторной работы в новой папке и не смешивайте с файлами предыдущих лабораторных работ. Архив содержит следующие файлы:

Файлы для редактирования:	
`models.py`	Модели персептрона и нейронных сетей для различных приложений.
`nn.py`	Мини-библиотека нейронных сетей.
Файлы, которые необходимо просмотреть	
`nn.py`	Мини-библиотека нейронных сетей.
`autograder.py`	Автооцениватель
`backend.py`	Бэкэнд-код для различных задач машинного обучения.
`data`	Наборы данных для классификации цифр и идентификации языка.

Ваш код будет автоматически проверяться автооценивателем. Поэтому не меняйте имена каких-либо функций или классов в коде, иначе вы внесете ошибку в работу автооценивателя.

6.4.3. Перед началом выполнения задания убедитесь, что у вас установлены библиотеки numpy, matplotlib и PyToch! Если Вы используете дистрибутив **conda**, то можно проверить список пакетов, установленных в текущей среде, набрав команду

conda list

Убедитесь, что в списке уставноленных пакетов имеются указанные выше пакеты.

6.4.4. Для решения задания 1 вам необходимо выполнить рекомендации, указанные ниже.

Дополните конструктор **init(self, dimensions)**. Он должен инициализировать веса персептрона. Убедитесь, что веса сохраняются в виде объекта **Parameter()** с

размерностью **1 x dimensions**. Это необходимо для того, чтобы автооцениватель, а также PyTorch, могли распознать вектор весов как параметр модели нейросети.

Реализуйте метод **run(self, x)**. Он должен вычислять скалярное произведение вектора весов и вектора входа, возвращая объект типа Tensor. Для вычисления скалярного произведения переменных, представленных тензорами, воспользуйтесь функцией PyTorch **tensordot(x,w,dims)**, где **dims** – кортеж из списков, указывающий индексы измерений вдоль которых вычиляется скалярное произведение.

Реализуйте метод **get_prediction(self, x)**, который должен возвращать 1, если скалярное произведение положительное и -1 – если отрицательное.

Запрограммируйте метод **train(self)**. Он должен многократно проходить по набору данных и обновлять параметры на примерах данных, которые были неправильно классифицированы. Когда проход по всему набору данных завершается без ошибок и достигается 100% точность обучения, обучение можно прекратить. Используйте алгоритм обучения бинарного персептрона, приведенный в п. 6.2.2.

Pytorch упрощает выполнение операций с тензорами. Обновление вектора весов пропорционально некоторому направлению, определяемому тензором **direction**, можно выполнить следующим образом:

self.w += direction * magnetic

Для этого задания, как и для всех остальных, каждый мини-блок данных, возвращаемый **DataLoader**, будет представляться в виде словаря: **{'x':features, 'label':label}**, где **label** — это метка, которая должна быть предсказана на основе признаков. Порядок работы с **DataLoader** изложен в п. 6.2.7.

Чтобы протестировать реализацию, выполните автооценивание:

python autograder.py -q q1

Автооценивателю необходимо не более 20 секунд для проверки решения. Если автооцениватель требует значительно большего времни то, вероятно, ваш код содержит ошибку.

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.5. В задании 2 вам необходимо реализовать методы, указанные ниже.

Реализуйте конструктор **RegressionModel.__init__** с необходимой инициализацией слоев нейросети. Используйте сеть не более, чем с двумя скрытыми слоями. Число нейронов в скрытом слое должно быть достаточно большим (100 – 200). Выходной слой должен быть линейным.

Реализуйте метод **RegressionModel.forward** который возвращает тензор размером **batch_size x 1** с предсказанными значениями функции sin(x). Метод просто выхывает слои описанные в конструкторе **RegressionModel.__init__.**

Реализуйте метод **RegressionModel.get_loss**, возвращаюший средний кварат для ошибки предсказанных значений функции sin(x). Этот метод должен вычислять предсказние **y_pred** с помощью **RegressionModel.forward** и вызывать функцию **mse_loss(y_pred, y)** для вычисления и возврата значений потерь.

Реализуйте метод **RegressionModel.train**, который выполнит обучение модели нейросети с использованием обновлений на основе градиента. Организация цикла обучения с использованием возможностей PyTorch изложена в п. 6.2.7.

Набор данных для этого задания содержит только обучающие данные. Для остановки обучения в качестве критерия используйте значение функции потерь. Ваша реализация задания должна обеспечить значение среднего квадрата ошибки обучения **train_loss** не выше 0.02, вычисляемого путем усреднения по всему набору обучающих данных. Для вычисления **train_loss** адаптируйте код:

```
data_x = torch.tensor(dataset.x,dtype=torch.float32)
labels = torch.tensor(dataset.y, dtype=torch.float32)
train_loss = model.get_loss(data_x, labels)
train_loss = train_loss.item()
```

Чтобы протестировать реализацию, выполните автооценивание:

```
python autograder.py -q q2
```

Обучение модели должно занять несколько минут.

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.6. В задании 3 необходимо завершить определение класса **DigitClassificationModel.** В целом структура кода этого класса аналогично заданию 2. Отличие состоит в том, что для задания 3, а также задания 4, в дополнение к обучающим данным, имеются валидационное и тестовое множества даных. Вам необходимо использовать **dataset.get_validation_accuracy()** для вычисления точности модели на вадидационных данных. Вычисленная точность необходима для принятия решения о прекращении обучения. Не используйте тестовый набор данных, он используется только во время автооценивания.

Ваша модель должна достичь точности не менее 97% на тестовом наборе данных. Обратите внимание, что автооцениватель оценивает точность модели на основе тестовых данных, в то время как у вас имеется доступ только к точности на валидационном множестве данных. Поэтому, если валидационная точность соответствует порогу 97%, вы все равно можете не пройти автооценивание. Рекомендуется установить немного более высокий порог остановки для валидационной точности, например, 97,5%.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q3
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.7. В задании 4 необходимо построить модель нейронной сети, которая определяет язык слова в наборе данных. На вход нейросети подаются отдельные символы слова: $\mathbf{x}_0$, $\mathbf{x}_1$,⋯ , $\mathbf{x}_{L-1}$, где L — длина слова. При этом каждый символ представляется “one-hot” бинарным вектором, содержащим 1 в позиции, соответсвующей номеру буквы в общем алфавите распознаваемых языков.

Начнем с применения начальной сети f_{initial}, которая похожа на сети из предыдущих заданий. Она принимает на свой вход символ $\mathbf{x}_0$ и вычисляет некоторый выходной вектор скрытого состояния $\mathbf{h}_1$ размерностью d:

$$\mathbf{h}_1 = f_{\text{initial}}(\mathbf{x}_0)$$

Далее мы объединим выход предыдущего шага со следующей буквой в слове, сгенерировав векторное представление первых двух букв слова. Для этого мы применим подсеть, которая принимает на вход новую букву и формирует новое скрытое состояние с учетом предыдущего скрытого состояния $\mathbf{h}_1$. Обозначим эту подсеть как f:

$$\mathbf{h}_2 = f(\mathbf{h}_1, \mathbf{x}_1)$$

Эти действия повторяем для всех букв входного слова. При этом скрытое состояние на каждом новом шаге аккумулирует представления всех букв, обработанных сетью на данный момент:

$$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_{i-1}) \text{ и т.д.}$$

Во всех этих вычислениях функция $f(\cdot,\cdot)$ соответствует одной и той же нейронной сети и использует те же обучаемые параметры, которые определены в f_{initial} . Таким образом, все параметры, используемые при обработке слов разной длины, являются общими.

Вы можете реализовать указанный процесс с помощью цикла **for**, перебирая элементы предоставляемого списка **xs** с кодами символов. На каждой итерации цикла вычисляет либо f_{initial}, либо f. Описанная выше сеть называется **рекуррентной нейронной сетью** (RNN) и изображена на рис.6.4 [11]. Здесь RNN используется для отображения слова «cat» в вектор скрытого состояния $\mathbf{h}_3$ фиксированной длины.

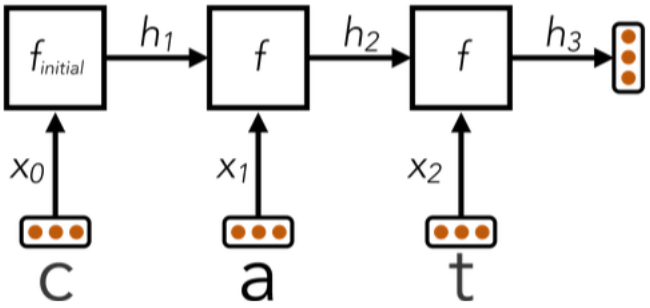


Рисунок 6.4 – Рекуррентная нейронная сеть

После того, как RNN обработает всю входную последовательность, она закодирует слово произвольной длины в виде вектора $\mathbf{h}_L$ фиксированного размера, где L — длина слова. Этот векторный эквивалент входного слова теперь можно обработать дополнительными выходными слоями для классификации языка слова.

Хотя приведенные выше выражения относятся к одному слову, на практике для эффективности необходимо использовать мини-блоки слов. Для упрощения решения задания, предоставляемый код гарантирует, что все слова в одном мини-блоке имеют одинаковую длину. В этом случае скрытое состояние $\mathbf{h}_i$ представляется матрицей $\mathbf{H}_i$ размерности **batch_size** x **d**.

При проектировании RNN учтите следующие рекомендации.

Начните с произвольной архитектуры $f_{\text{initial}}(x)$, аналогичной предыдущим заданиям, при условии, что она имеет хотя бы одну нелинейность.

Рекомендуется использовать следующий подход построения сети $f(\cdot,\cdot)$. Первый слой начальной подсети f_{initial} должен обеспечивать умножение вектора $\mathbf{x}_0$ на матрицу весов $\mathbf{W}_x$ для получения $\mathbf{h}_1 = f(\mathbf{x}_0 \cdot \mathbf{W}_x)$ Для обработки последующих букв следует заменить это вычисление на $\mathbf{h}_i = f(\mathbf{x}_i \cdot \mathbf{W}_x + \mathbf{h}_{i-1} \cdot \mathbf{W}_h)$. Рекомендуется в качестве нелинейности f использовать функцию ReLu. В этом случае вам следует заменить вычисление первого шага **h=relu(self.lin_layer1(xs[0]))** на вычисление вида **h=relu(self.lin_layer1(x) + self.lin_layer2(h))**, где слои **lin_layer1** и **lin_layer2** реализуют умножение векторов **x** и **h** на матрицы весов $\mathbf{W}_x$ и $\mathbf{W}_h$, соответственно.

Скрытый размер d должен быть достаточно большим. Цикл обучения реализуется аналогично заданиям 2 и 3. В качестве функции потерь используйте **cross_entropy(y_pred,y).**

Начните с неглубокой сети f и определите подходящие значения d для вектора скрытого состояния и скорости обучения, прежде чем делать сеть глубже. Если вы сразу начнете с глубокой сети, у вас будет экспоненциально больше комбинаций гиперпараметров, и любой неправильный гиперпараметр может привести к резкому снижению эффективности сети.

Для решения задания вам необходимо завершить реализацию класса **LanguageIDModel.** После обучения нейросеть должна достичь точности не ниже 81% на тестовом наборе данных.

Набор данных задания был создан с помощью автоматизированной обработки текста. Он может содержать ошибки. Тем не менее, эталонная реализация нейросети может правильно классифицировать более 89% слов тестового набора.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

python autograder.py -q q4

Внесите код разработанных методов и результаты тестирования в отчет.

6.4.8. В задании 5 необходимо разработать и обучить сверточную нейронную сеть. Рассмотрим основы построения сверточных слоев.

Часто при обучении нейронной сети возникает необходимость использовать более сложные слои, чем простые линейные слои. Одним из распространенных типов слоев является сверточный слой. Сверточные слои позволяют лучше учесть пространственную информацию, содержащуюся в многомерных входных данных. Например, рассмотрим следующие входные данные, представляющие изображение в виде двумерной матрицы:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{d1} & x_{d2} & \dots & x_{dn} \end{bmatrix}.$$

Если бы мы использовали линейный слой, подобный тому, который использовался в задании 2, то для подачи этих данных на вход нейронной сети нам пришлось бы преобразовать их в следующую форму:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} & \dots & x_{dn} \end{bmatrix}.$$

Но в задачах классификация изображений лучше сохранить пространственную информацию, содержащуюся в исходных данных, и обрабатывать их в матричной форме. Для этого применяют **сверточные слои (**convolutional layers). Двумерный сверточный слой хранит веса нейронов в виде двумерной матрицы, например:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}.$$

При подаче входных данных слой выполняет свертку матрицы входа и матрицы весов. После выполнения этого **сверточная нейронная сеть** (CNN – Convolutional Neural Network) может преобразовать выход сверточного слоя к одномерному вектору и передать его полносвязным слоям для дальнейшей обработки.

Выходные данные двумерной свертки можно следующим образом:

$$\mathbf{Y} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ y_{d1} & y_{d2} & \cdots & y_{dn} \end{bmatrix},$$

где любой элемент y_{ij} получается в результате поэлементного умножения матрицы **W** и части входной матрицы **X**, которая начинается с элемента x_{ij} и имеет ту же ширину и высоту, что и матрица **W**. Затем все элементы промежуточной матрицы произведений суммируются для вычисления y_{ij}. Например, для нахождения значения y_{22} необходимо поэлементно умножить **W** на следующую часть матрицы входа:

$$\begin{bmatrix} x_{22} & x_{23} \\ x_{32} & x_{33} \end{bmatrix}.$$

В реультате получим матрицу произведений

$$\begin{bmatrix} x_{22} * w_{11} & x_{23} * w_{12} \\ x_{32} * w_{21} & x_{33} * w_{22} \end{bmatrix}.$$

Выполнив суммирование всех элементов этой матрицы, вычислим y_{22}:

$$y_{22} = x_{22} * w_{11} + x_{23} * w_{12} + x_{32} * w_{21} + x_{33} * w_{22}$$

Иногда при применении свертки входная матрица дополняется по краям нулями, чтобы размер выходной матрицы сверточного слоя совпадал с размером входной матрицы. В этом задании вам это не требуется делать. Поэтому ваша выходная матрица должна быть меньше входной матрицы. В общем случае для вычисления размеров выходной матрицы используются варажения:

Ht = 1 + (H + 2 * pad - HH) / stride
Wt = 1 + (W + 2 * pad - WW) / stride

Здесь **H, W** – высота и ширина матрицы входных данных; **Ht, Wt** – высота и ширина матрицы выходных данных; **HH, WW** – высота и ширина окна (матрицы весов) свертки; **stride** – шаг смещения окна свертки (обычно 1) при его наложении на мат-

рицу входных данных; **pad** – число строк или столбцов добаленных нулей к исходной матрице данных (в лабораторной работе 0). Обратите внимание, что размеры выходной матрицы данных должны быть целыми значениями.

В этом задании вам необходимо сначала дописать функцию **Convolve** в **models.py**. Эта функция принимает входную матрицу и матрицу весов и вычисляет их свертку. Обратите внимание, что входная матрица всегда больше матрицы весов и всегда будет передаваться по одной за раз, поэтому вам не нужно выполнять свертку нескольких входных матриц одновременно. При вычислении каждого элемента y_{ij} выходной матрицы вы должны накладывать левый верхний угол окна свертки на соответствующую позицию входной матрицы, вырезать из нее подматрицу **x_slice**

```
x_slice = input[i:(i + HH), j:(j + WW)]
```

и проводить вычисление свертки путем вычисления суммы поэлентных произведений **x_slice** и матрицы весов **weight** (см. выше):

```
y[i, j] = tensordot(x_slice,weight,dims=2)
```

После определения функции **Convolve** завершите определение класса **DigitConvolutionalModel()** в **models.py**. Можно повторно использовать большую часть кода задания 3.

Автооцениватель сначала проверит функцию **Convolve**, чтобы убедиться, что она правильно вычисляет свертку двух матриц. Затем он протестирует вашу модель, чтобы увидеть, может ли она достичь точности 80% на упрощенном подмножестве данных MNIST. Поскольку это задание касается проверки функции **Convolve()**, которую вы написали, то модель должна обучаться относительно быстро.

Сверточная сеть, разработанная в этом задании, скорее всего, будет работать медленнее по сравнению со сверточными слоями, предоставляемыми PyTorch. Это ожидаемо, так как пакеты вроде Pytorch используют оптимизацию для ускорения вычисления сверток.

В методе **run()** мы уже реализовали вызов сверточного слоя и преобразование его матричного выхода в вектор. Далее вам необходимо обработать этот вектор линейными слоями, аналогично заданию 3. Вам понадобится всего пара слоев, чтобы достичь точности 80%.

Чтобы протестировать вашу реализацию, запустите автооцениватель:

```
python autograder.py -q q5
```

Внесите код разработанных методов и результаты тестирования в отчет.

6.5. Содержание отчета

Цель работы, структурные схемы исследуемых нейросетей в соответствии с заданиями 1-5, описание используемых данных и функций потерь, код реализован-

ных функций м методов с комментариями в соответствии с заданиями 1-5, результаты автооценивания заданий, выводы по проведенным экспериментам с разными нейронными сетями.

6.6. Контрольные вопросы

6.6.1 Назовите виды алгоритмов обучения? Объясните, что понимают под обучением с учителем? Что понимают под обучением без учителя?

6.6.2. На какие подмножества разбивают наборы данных, используемые при разработке нейросетей? Назовите назначение каждого из подмножеств?

6.6.3. Что такое гиперпарамеры алгоритма обучения?

6.6.4. Нарисуйте схему бинарного персептрона и объясните алгоритм его обучения? Что представляет собой граница решения бинарного персептрона и как её построить?

6.6.5. Запишите модель прогнозирования линейной регрессии. Как оценивается качество модели линейной регрессии?

6.6.6. Объясните алгоритм градиентного спуска? Сформулируйте отличия стохастического градентного спуска и мини-блочного градиентного спуска?

6.6.7. Запишите выражения логистической регрессии? Запишите выражения многоклассовой логистической регрессии? Что такое softmax? Что называют логитами? Запишите выражение функции потерь, которую называют кросс-энтропией? Запишите выражение бинарной кросс-энтропии?

6.6.8. Нарисуйте схему формального нейрона и запишите выражение для его выхода? Какие активионные функции вам известны? Постройте их графики и запишите выражения?

6.6.9. Нарисуйте схему многослойной полносязной сети прямого распространения и запишите уравнения прямого распространения данных? Что называют скрытым слоем? Как обозначают структуру нейронной сети с помощью символов?

6.6.10. Охарактеризуйте назначение пакета Pytorch? Что такое тензор в PyTorch? Как его определить? Объясните назначение, определение и использование класса **Linear**? Объясните функции **movedim(), cross_entropy(), mse_loss()**? Объясните как с помощью объекта **DataLoader** обеспечить загрузку данных и выборку мини-блоков?

6.6.11. Напишите на Pytoch типовой цикл обучения простейшей 2-х слойной нейросети? Как в Pyrtoch осуществляется вычисление градиентов и применение их обновления параметров нейросети?

6.6.12. Нарисуйте структуру простейшей рекуррентной сети и объясните выражения, лежащие в основе ее функционирования?

6.6.13. Нарисуйте и объясните архитектуру RNN для распознавния языка?

6.6.14. Обясните принцип построения сверточного слоя и алгоритм вычисления свертки?