Конспект по методам ML/DL для собеседования (Middle ML/DS)

Содержание

1 Регрессия и регуляризация

1.1 Линейная регрессия

Определение: Линейная регрессия – это статистический метод, моделирующий зависимость непрерывной целевой переменной от входных признаков линейной комбинацией этих признаков.

Интуиция: Модель пытается провести плоскость (или гиперплоскость) в пространстве признаков, чтобы максимально приблизить ей исходные данные. Каждому признаку сопоставляется вес (коэффициент), а прогноз = сумма взвешенных признаков + смещение.

Применение: Задачи регрессии (прогнозирование цены, прогноз спроса, экономики и т.д.), когда предполагается линейная зависимость. Широко используется для быстрой аппроксимации данных.

Плюсы:

- Простота реализации и интерпретации.
- Быстрый расчёт, аналитическое решение (OLS).
- Может служить базовой моделью для линейной зависимости.

Минусы:

- Плохо моделирует сложные (нелинейные) зависимости.
- Чувствительна к выбросам.
- Может переобучаться (если много параметров), особенно при малом числе объектов.
- Неустойчива к мультиколлинеарности.

Пример (sklearn):

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

```

Сравнение: В отличие от полиномиальной регрессии (которая вводит нелинейные признаки), чистая линейная регрессия не добавляет новых признаков. Если зависимость нелинейна, можно расширить модель (полиномиальные признаки) или перейти к более сложным методам.

1.2 Lasso (L1-регуляризация)

Определение: Lasso (Least Absolute Shrinkage and Selection Operator) – это вариант линейной регрессии с L1-регуляризацией. В задачу обучения добавляется штраф пропорциональный абсолютным значениям коэффициентов модели.

Интуиция: L1-пенализация заставляет многие коэффициенты прижиматься к нулю ("разреживает" модель), фактически выбирая наиболее значимые признаки. Это как обычная регрессия, но со «штрафом» за большие веса.

Применение: Когда данных много и важна автоматическая селекция признаков (например, геномика, текстовые данные, где много признаков). Служит для отбора признаков и уменьшения переобучения.

Плюсы:

- Даёт разреженное решение (обнуляет неважные признаки), что улучшает интерпретируемость и устойчивость к шуму.
- Снижает переобучение и помогает при избытке признаков.

Минусы:

- Может быть нестабильна при высокой коррелированности признаков (обычно выбирает один из группы коррелированных).
- Жёсткая регуляризация может привести к сильным смещениям оценок.

Пример (sklearn):

```
from sklearn.linear_model import Lasso
# ИСПРАВЛЕНО: заменено тире на дефис
model = Lasso(alpha=1.0) # alpha - коэффициент регуляризации
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

```

Сравнение: В отличие от Ridge (L2-регуляризация), Lasso даёт жёсткое обнуление некоторых коэффициентов. ElasticNet сочетает L1 и L2, чтобы компенсировать слабые стороны Lasso.

1.3 Ridge (L2-регуляризация)

Определение: Ridge-регрессия – это линейная регрессия с L2-пеналью (квадрат суммы коэффициентов). Регуляризация корректирует переобучение и мультиколлинеарность.

Интуиция: L2-пенализация «тянет» все коэффициенты к нулю плавно, но не делает их точно нулевыми. Модель остаётся линейной, но со сжатыми весами, что улучшает обобщение.

Применение: Аналогично линейной регрессии, но когда нужно бороться с переобучением и шкалировать/стабилизировать веса. Особенно полезно при большом числе признаков и их высокой коллинеарности.

Плюсы:

- Снижает дисперсию модели, стабильнее в случае коррелированных признаков (все коэффициенты уменьшаются, а не отбираются один).
- Легко решается численно.

Минусы:

- Не даёт разреженности (все признаки остаются с ненулевыми весами).
- Неявно вводит смещение (сдвиг предсказания).
- Не защищает от выбросов.

Пример (sklearn):

```
from sklearn.linear_model import Ridge
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

```

Сравнение: По сравнению с Lasso (L1), Ridge более равномерно «уменьшает» коэффициенты. ElasticNet сочетает их, Lasso отбирает признаки, Ridge – нет.

1.4 Elastic Net (L1+L2-регуляризация)

Определение: Elastic Net – это линейная регрессия с комбинацией L1- и L2-пеналей. Формально, штраф $= \alpha \cdot L1 + \beta \cdot L2$.

Интуиция: Сочетает свойства Lasso и Ridge: часть признаков обнуляется, другие — сжимаются, что улучшает качество, особенно при высоко коррелированных признаках.

Применение: Используется при большом числе признаков, когда есть коррелированные группы признаков. Часто даёт более точные и стабильные результаты, чем чистый Lasso или Ridge.

Плюсы:

- Обеспечивает баланс между разреженностью и гладкостью решения.
- Группы коррелированных признаков обычно обрабатываются лучше, чем Lasso.

Минусы:

- Требует настройки двух гиперпараметров (доли L1 и L2).
- Аналитическая трактовка сложнее.
- Может давать промежуточное решение, не столь интерпретируемое, как чистый Lasso.

Пример (sklearn):

```
from sklearn.linear_model import ElasticNet
model = ElasticNet(alpha=1.0, l1_ratio=0.5)
# l1_ratio = доля L1-пенали
model.fit(X_train, y_train)

```

Сравнение: Гибрид Lasso и Ridge. Если l1_ratio=1 → Lasso, l1_ratio=0 → Ridge. Обычно лучше, чем оба по отдельности.

2 Деревья решений и ансамбли

2.1 Decision Tree (Дерево решений)

Определение: Дерево решений – это непараметрический алгоритм, строящий модель в виде дерева, где каждый узел – тест по признаку, а листья – ответ (класс или значение). Подходит для классификации и регрессии.

Интуиция: Дерево «разбивает» пространство признаков на части с помощью условий вида "$x_i > threshold$", пока каждая часть не станет однородной. Это иерархическая модель принятия решений (root → ветви → листья).

Применение: Классификация (лицензирование, прогноз отказов, определение категории) или регрессия (например, предсказание цены дома). Хороша, когда нужны понятные правила принятия решений.

Плюсы:

- Легко интерпретировать и визуализировать.
- Не требует масштабирования признаков.
- Автоматически захватывает нелинейные зависимости и взаимодействия признаков.

- Работает с данными разной природы (числовые/категориальные).

Минусы:

- Склонна к переобучению (особенно глубокие деревья) – на тренировочных данных даёт 100% точность, но плохо обобщает.
- Чувствительна к шуму и малым изменениям данных (неустойчивость).
- Может создавать очень сложные (и плохо читаемые) правила.

Пример (sklearn):

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(max_depth=5)
model.fit(X_train, y_train)

```

Сравнение: По сравнению с случайным лесом – одиночное дерево проще и быстрее обучается, но менее точно и более подвержено переобучению.

2.2 Random Forest (Случайный лес)

Определение: Random Forest – это ансамбль (сборник) решений, состоящий из множества деревьев, выводы которых объединяются (например, голосованием). Каждый объект классифицируется «средним» ответом деревьев (или усредняем в регрессии).

Интуиция: Берём много разных деревьев (каждое обучается на случайной подвыборке данных и случайном наборе признаков). Поскольку деревья склонны к переобучению, их «усреднение» значительно уменьшает разброс (дисперсию) итоговой модели.

Применение: Как для классификации, так и регрессии. Широко используется в реальных задачах за счёт простоты и высокой точности (финансовый скоринг, компьютерное зрение, здравоохранение и др.).

Плюсы:

- Превосходит одиночные деревья по точности и устойчивости.
- Эффективно борется с переобучением.
- Может работать с большими наборами данных.
- Оценивает важность признаков (feature importance).

Минусы:

- Теряет интерпретируемость (чтение сотен деревьев бессмысленно).
- Обучается и предсказывает медленнее, чем одно дерево.
- Чувствителен к выбросам, хотя не так сильно, как одно дерево.

Пример (sklearn):

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, max_depth=10)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

```

Сравнение: По сравнению с бэггингом – Random Forest является частным случаем бэггинга деревьев с добавлением случайности выбора признаков.

2.3 Bagging (бэггинг, Bootstrap Aggregation)

Определение: Бэггинг – это метод ансамблирования, при котором множество базовых моделей обучаются независимо на разных бутстрэп-выборках исходных данных, а их ответы усредняются.

Интуиция: Каждый базовый алгоритм (обычно слабый, например, дерево) видит случайный поднабор объектов (с повторениями). Усреднение (или голосование) результатов разных моделей снижает дисперсию итоговой модели.

Применение: Уменьшение переобучения. В основном используется с деревьями решений (что и даёт Random Forest), но применимо и к любым другим моделям.

Плюсы:

- Эффективно уменьшает разброс модели, повышает стабильность и точность ансамбля.
- Прост в понимании и реализации.

Минусы:

- Требует больше вычислений (несколько моделей вместо одной).
- Модели внутри не «сотрудничают», поэтому бэггинг не сильно снижает смещение (bias), а только разброс.

Пример (sklearn):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
base = DecisionTreeClassifier()
model = BaggingClassifier(base_estimator=base, n_estimators=10)
model.fit(X_train, y_train)

```

Сравнение: В отличие от бустинга, бэггинг обучает модели параллельно и одинаково взвешивает их. В бустинге модели обучаются последовательно и нацелены исправлять ошибки друг друга.

2.4 Boosting (бустинг)

Определение: Бустинг – это метод ансамблирования, который последовательно обучает слабые модели, каждой новой модели пытаясь исправить ошибки предыдущих. В результате получается сильный ансамбль.

Интуиция: Начинаем с простой модели (например, небольшого дерева). Затем обучаем следующую модель на данных, где «важнее» объекты, плохо предсказанные предыдущей (например, увеличивая их вес). Повторяем несколько раз. Итоговый прогноз – комбинированный (обычно взвешенное голосование) ответ всех моделей.

Применение: Классификация и регрессия. Часто использует деревья небольшой глубины (решающие «слабые» деревья). Хорошо работает, когда нужен максимально точный классификатор.

Плюсы:

- Резко снижает смещение (находит сложные зависимости), часто даёт очень высокую точность.
- Адаптивно фокусируется на трудных примерах.

- Хорошая обобщающая способность при правильной настройке.

Минусы:

- Риск переобучения, особенно если слишком много итераций или слабые модели слишком сложные.
- Более дорого по времени обучения (последовательный процесс).
- Требует тонкой настройки гиперпараметров (скорость обучения, количество итераций).

Пример (sklearn AdaBoost):

```
from sklearn.ensemble import AdaBoostClassifier
model = AdaBoostClassifier(n_estimators=50, learning_rate=1.0)
model.fit(X_train, y_train)

```

Сравнение: В отличие от бэггинга (параллельная обработка, уменьшает разброс), бустинг обучается последовательно и уменьшает смещение, фокусируясь на «ошибках» предыдущих моделей.

2.5 Gradient Boosting

Определение: Специальный вариант бустинга, где каждая новая модель обучается на градиентах (остатках) предыдущей – минимизируя функционал потерь через метод градиентного спуска.

Интуиция: На каждой итерации строим дерево не просто на ошибках, а на градиенте потерь (разности между предсказанием и реальным значением). Суммируя предсказания последовательных деревьев, постепенно минимизируем ошибку.

Пример (sklearn):

```
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1)
model.fit(X_train, y_train)

```

Сравнение: Это алгоритмический подход бустинга; AdaBoost основывается на весах обучающих примеров, а GradientBoosting – на подгонке остатков (градиента).

2.6 XGBoost

Определение: XGBoost (Extreme Gradient Boosting) – это высокопроизводительная реализация градиентного бустинга, оптимизированная по скорости и масштабу.

Интуиция: Строит деревья последовательным бустингом, использует параллельное обучение и регуляризацию, эффективно работает с пропущенными значениями.

Применение: Используется как drop-in замена GradientBoosting. Известен в соревнованиях ML за высокую точность и гибкость.

Плюсы:

- Очень быстрый за счёт оптимизации и поддерживает параллельные вычисления.
- Встроенные методы регуляризации (L1, L2) позволяют бороться с переобучением.
- Автоматически обрабатывает пропуски.

Минусы:

- Требует установки внешней библиотеки.
- Может быть сложен в настройке (множество гиперпараметров).

Пример (XGBoost):

```
import xgboost as xgb
model = xgb.XGBClassifier(n_estimators=100, learning_rate=0.1)
model.fit(X_train, y_train)

```

Сравнение: XGBoost – это именно реализация градиентного бустинга. По сравнению с обычным sklearn.GradientBoosting значительно быстрее на больших данных.

2.7 LightGBM

Определение: LightGBM – это фреймворк градиентного бустинга, использующий особенности типа tree-based learning (например, обучает по листьям). Разработан Microsoft для повышения эффективности.

Интуиция: Вместо традиционного построения уровневого дерева (как в XGBoost), LightGBM строит дерево по принципу «листья-любитель», что даёт быстрее скорость обучения и более качественные деревья на больших данных.

Применение: Крупные датасеты и задачи с большим числом признаков. Часто выбирается, если нужны очень быстрые итерации и высокая точность при больших объемах.

Плюсы:

- Очень высокая скорость обучения и малый расход памяти.
- Поддерживает категориальные признаки «из коробки».
- Обычно работает быстрее XGBoost на большом объёме данных.

Минусы:

- Иногда переобучается (leaf-wise строит глубокие ветви).
- Может хуже работать на малых данных или сильно несбалансированных.

Пример (LightGBM):

```
import lightgbm as lgb
model = lgb.LGBMClassifier(n_estimators=100, learning_rate=0.1)
model.fit(X_train, y_train)

```

2.8 CatBoost

Определение: CatBoost – это алгоритм градиентного бустинга от компании Yandex. Специально разработан для эффективной работы с категориальными признаками.

Интуиция: Наряду с обычными деревьями, CatBoost использует алгоритмы обработки категорий (например, порядковое кодирование при обучении), чтобы избегать необходимости ручного one-hot-кодирования.

Применение: Задачи с большим числом категориальных признаков. Часто даёт лучшие результаты «из коробки» без долгой настройки.

Плюсы:

- Высокая точность, особенно на смешанных данных с категориями.

- Встроенно обрабатывает категориальные признаки.
- Меньше риск переобучения.

Минусы:

- Библиотеку необходимо устанавливать отдельно.
- Обучение может быть менее параллельным, чем LightGBM.

Пример (CatBoost):

```
from catboost import CatBoostClassifier
model = CatBoostClassifier(iterations=100, learning_rate=0.1)
model.fit(X_train, y_train, cat_features=[...])
# передаются индексы категориальных колонок

```

Сравнение: Часто превосходит XGBoost и LightGBM по точности на задачах с категориальными признаками.

3 Кластеризация

3.1 K-Means

Определение: K-means – алгоритм кластеризации, цель которого разбить n объектов на k кластеров, минимизируя сумму квадратов расстояний объектов до центроидов своих кластеров.

Интуиция: Инициализируем k центроидов, затем итеративно: 1) присваиваем каждый объект к ближайшему центру, 2) пересчитываем центроиды как средние значений объектов в кластере, 3) повторяем до сходимости.

Применение: Сегментация клиентов, кластеризация документов, сжатие изображений. Подходит, когда кластеры примерно сферические.

Плюсы:

- Простота и понятность, быстрый алгоритм.
- Хорошо масштабируется на большие данные.

Минусы:

- Нужно заранее задать k.
- Чувствителен к начальной инициализации.
- Плохо работает с выбросами и кластерами неравного размера или формы.

Пример (sklearn):

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3, init='k-means++', n_init=10)
labels = model.fit_predict(X)

```

Сравнение: В отличие от DBSCAN, K-means требует заранее число кластеров и не находит шум; лучше работает при однородных по плотности кластерах.

3.2 DBSCAN

Определение: DBSCAN (Density-Based Spatial Clustering) – метод кластеризации на основе плотности. Группирует вместе «плотные» участки данных, а разреженные области помечает как шум.

Интуиция: Основные параметры – ε (eps) и MinPts. Если вокруг точки в радиусе ε находится $\geq$ MinPts соседей, точка считается «ядром» кластера. Кластеры строятся рекурсивно от ядер.

Применение: Когда кластеры имеют произвольную форму или неизвестно их число (геоданные, аномалии).

Плюсы:

- Не требует заранее k.
- Выявляет кластеры любых форм, отмечает выбросы.

Минусы:

- Выбор параметров ε и MinPts критичен.
- Плохо справляется с кластерами разной плотности.

Пример (sklearn):

```
from sklearn.cluster import DBSCAN
model = DBSCAN(eps=0.5, min_samples=5)
labels = model.fit_predict(X)

```

Сравнение: В отличие от K-Means, DBSCAN не требует k и может находить «извилистые» кластеры, но чувствителен к выбору ε.

4 Классификация

4.1 Support Vector Machine (SVM)

Определение: SVM – это алгоритм классификации, который ищет оптимальную гиперплоскость, разделяющую классы с максимальным зазором (margin).

Интуиция: SVM стремится найти границу, при которой расстояние от ближайших объектов разных классов до этой границы максимально (max-margin). Используется kernel-трюк для нелинейного разделения.

Применение: Текстовые данные, изображения, где размерность может быть большой.

Плюсы:

- Эффективен в пространствах большой размерности.
- Использует поддержку векторов, что экономит память.
- Гибкий благодаря разным ядрам.

Минусы:

- Сложность обучения от $O(n^2)$ до $O(n^3)$, плохо масштабируется.
- Чувствителен к шуму.
- Не даёт по умолчанию вероятностных предсказаний.

Пример (sklearn):

```
from sklearn.svm import SVC
model = SVC(kernel='rbf', C=1.0, probability=True)
model.fit(X_train, y_train)

```

Сравнение: В отличие от kNN, SVM строит глобальную границу и может лучше обобщать.

4.2 Naive Bayes (Наивный Байес)

Определение: Семейство вероятностных классификаторов, базирующихся на теореме Байеса с «наивным» предположением о независимости признаков.

Интуиция: Оценивает вероятности $P(\text{класс}|\text{признаки}) \propto P(\text{признаки}|\text{класс}) \cdot P(\text{класс})$.

Применение: Текстовая классификация (спам-фильтры), диагностика заболеваний.

Плюсы:

- Прост и быстр в обучении.
- Хорошо работает при небольших выборках и многоклассовых задачах.

Минусы:

- Сильно зависит от предположения независимости признаков.
- Чувствителен к нулевым вероятностям.

Пример (sklearn):

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, y_train)

```

Сравнение: По сравнению с SVM, наивный Байес проще и быстрее, но обычно менее точен при большом числе зависимых признаков.

4.3 KNN (K-ближайших соседей)

Определение: Простой алгоритм, где класс точки определяется по большинству из k ближайших ей объектов. Относится к ленивым методам.

Интуиция: Для нового объекта ищутся k самых близких объектов обучающей выборки. Происходит «голосование».

Применение: Часто используется как эталон или при отсутствии времени на обучение.

Плюсы:

- Простота реализации (нет фазы обучения).
- Адаптивность при добавлении новых данных.

Минусы:

- Медленный на больших выборках.
- Чувствителен к масштабу признаков и «проклятию размерности».

Пример (sklearn):

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

```

Сравнение: В отличие от линейных моделей, KNN не строит явной разделяющей поверхности.

5 Снижение размерности

5.1 PCA (Principal Component Analysis)

Определение: Метод линейного снижения размерности, который проецирует данные в новое пространство признаков (главные компоненты), сохраняя максимальную дисперсию.

Интуиция: Ищутся направления, вдоль которых данные «растянуты» сильнее всего.

Применение: Уменьшение числа признаков, визуализация, удаление линейной корреляции.

Плюсы:

- Снижает размерность при минимальной потере информации.
- Устраняет линейную зависимость между признаками.

Минусы:

- Учитывает только линейные связи.
- Сложно интерпретировать новые признаки.

Пример (sklearn):

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

```

Сравнение: По сравнению с LDA, PCA не учитывает метки классов.

5.2 LDA (Linear Discriminant Analysis)

Определение: Метод снижения размерности для разделения классов. Ищет линейные комбинации признаков, которые максимально разделяют классы.

Интуиция: Максимизирует отношение между-классовой и внутри-классовой дисперсий.

Применение: Обычно перед классификатором.

Плюсы:

- Учитывает метки классов.
- Часто повышает точность классификации.

Минусы:

- Предполагает нормальное распределение признаков.
- Неустойчива к выбросам.

Пример (sklearn):

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X, y)

```

Сравнение: В отличие от PCA, LDA – супервизированный метод.

5.3 NMF (Non-Negative Matrix Factorization)

Определение: Метод факторизации матриц, который разлагает матрицу данных на две матрицы с неотрицательными элементами.

Применение: Тематическое моделирование, разложение изображений.

Плюсы:

- Даёт интерпретируемые компоненты (нет отрицаний).

Минусы:

- Работает только с неотрицательными данными.

Пример (sklearn):

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=5, init='random', random_state=0)
W = nmf.fit_transform(X)
H = nmf.components_

```

6 Обработка категориальных признаков

6.1 One-Hot Encoding (OHE)

Определение: Метод представления категориальных переменных в двоичном формате. Для каждой категории создаётся отдельный бинарный признак.

Интуиция: Переводим номинальные признаки в числовой вид без ввода ложного порядка.

Плюсы:

- Простота и «честность» представления.

Минусы:

- Резкий рост размерности.
- Разреженность данных.

Пример (pandas/sklearn):

```
# С помощью pandas:
X_encoded = pandas.get_dummies(df, columns=['category_feature'])
# Или sklearn:
from sklearn.preprocessing import OneHotEncoder
encoder = OneHotEncoder(sparse=False)
X_enc = encoder.fit_transform(df[['category_feature']])

```

7 Глубокое обучение

7.1 Autoencoder (Автоэнкодер)

Определение: Тип нейронной сети, обучаемый без учителя, который кодирует входные данные в компактное представление, а затем пытается восстановить исходные данные.

Интуиция: Сеть учится «сжимать» данные (энкодер) и восстанавливать (декодер). Латентное пространство содержит самую важную информацию.

Применение: Уменьшение размерности, детектирование аномалий, удаление шума.

Плюсы:

- Захватывает нелинейные зависимости.
- Обучается без меток (self-supervised).

Минусы:

- Требует подбора архитектуры.
- Результат трудно интерпретировать.

Пример (Keras):

```
from keras.models import Model
from keras.layers import Input, Dense
input_dim = X_train.shape[1]
encoding_dim = 32
input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256)

```

7.2 Dropout

Определение: Приём регуляризации, при котором на каждом шаге обучения случайно «выключается» часть нейронов.

Интуиция: Снижается зависимость от одного узла, похоже на ансамблирование.

Применение: В глубоких сетях для борьбы с переобучением.

Плюсы:

- Эффективно борется с переобучением.
- Простота использования.

Минусы:

- Увеличивает время обучения.
- Может недообучить модель.

Пример (Keras):

```
from keras.layers import Dense, Dropout
model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(input_dim,)))
# ИСПРАВЛЕНО: заменены кавычки-елочки на обычные
model.add(Dropout(0.5)) # 50% нейронов случайно "выключены"
model.add(Dense(10, activation='softmax'))

```

7.3 RNN (Recurrent Neural Network)

Определение: Рекуррентная нейронная сеть для последовательных данных. Имеет «память» (состояния предыдущих шагов).

Интуиция: На каждом шаге входом являются текущие признаки и скрытое состояние из предыдущего шага.

Применение: Тексты, аудио, видео, временные ряды.

Плюсы:

- Учитывает порядок и контекст.
- Применим к переменной длине входа.

Минусы:

- Проблема исчезающего градиента.
- Обучение последовательное (медленное).

Пример (Keras):

```
from keras.models import Sequential
from keras.layers import SimpleRNN, LSTM
model = Sequential()
model.add(SimpleRNN(64, input_shape=(timesteps, features)))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')

```

8 Методы оптимизации (Optimizers)

Математическое обоснование: Цель обучения нейронной сети — найти набор параметров (весов) θ, который минимизирует функцию потерь $J(\theta)$. Оптимизаторы — это итеративные методы, которые пытаются найти минимум этой функции. Основой большинства оптимизаторов является метод градиентного спуска, который обновляет параметры в направлении, противоположном градиенту $\nabla J(\theta)$, так как градиент указывает на направление наискорейшего роста функции.

8.1 SGD (Stochastic Gradient Descent)

Математическое обоснование: Классический градиентный спуск вычисляет градиент по всему набору данных, что вычислительно дорого. SGD аппроксимирует этот градиент, вычисляя его на небольшом случайном подмножестве данных (мини-батче). Хотя градиент для мини-батча является "шумной"оценкой истинного градиента, в среднем он указывает в правильном направлении. Этот "шум"также помогает алгоритму избегать "мелких"локальных минимумов.

Формула обновления:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t; x^{(i:i+n)}; y^{(i:i+n)})$$

где η — скорость обучения (learning rate), а градиент вычисляется по мини-батчу размером n.

Параметры в коде (PyTorch):

```
    import torch.optim as optim
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0)
    # lr: скорость обучения (float)
    # momentum: (рассмотрен далее)

```

8.2 Momentum

Математическое обоснование: Momentum вводит "вектор скорости"v_t, который является экспоненциально взвешенным скользящим средним прошлых градиентов. Это помогает ускорить сходимость в направлениях с постоянным градиентом и сгладить осцилляции в направлениях, где знак градиента часто меняется. Это аналог инерции в физике: объект продолжает двигаться в том же направлении, сохраняя скорость.

Формула обновления:

$$v_{t+1} = \gamma v_t + \eta \cdot \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

где γ — коэффициент моментума, сохраняющий долю предыдущего обновления.

Параметры в коде (PyTorch):

```
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
    # momentum: коэффициент моментума (float), обычно ~0.9

```

8.3 AdaGrad (Adaptive Gradient Algorithm)

Математическое обоснование: AdaGrad адаптирует скорость обучения для каждого параметра индивидуально. Он делит глобальную скорость обучения на корень из суммы квадратов прошлых градиентов для этого параметра. Таким образом, для параметров, которые уже претерпели большие изменения (имеют большие накопленные градиенты), скорость обучения уменьшается. Это особенно эффективно для разреженных данных, где редкие признаки требуют больших обновлений, а частые — меньших.

Формула обновления:

$$G_t = G_{t-1} + (\nabla J(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

где G_t — диагональная матрица, где каждый элемент i, i — это сумма квадратов градиентов по параметру θ_i, ϵ — малая константа для стабильности, а $\odot$ — поэлементное умножение.

Параметры в коде (PyTorch):

```
    optimizer = optim.Adagrad(model.parameters(), lr=0.01, eps=1e-8)
    # lr: начальная скорость обучения (float)
    # eps: член для улучшения численной стабильности (float)

```

8.4 RMSprop (Root Mean Square Propagation)

Математическое обоснование: RMSprop модифицирует AdaGrad, чтобы решить проблему монотонного и быстрого затухания скорости обучения. Вместо накопления всех прошлых квадратов градиентов, RMSprop использует экспоненциально взвешенное скользящее среднее. Это позволяет "забывать"старые градиенты, предотвращая слишком сильное падение скорости обучения.

Формула обновления:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)(\nabla J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot \nabla J(\theta_t)$$

где $E[g^2]_t$ — скользящее среднее, γ — коэффициент затухания.

Параметры в коде (PyTorch):

```
optimizer = optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99, eps=1e-8)
# alpha: коэффициент сглаживания (float), аналог gamma
# eps: член для улучшения численной стабильности (float)

```

8.5 Adam (Adaptive Moment Estimation)

Математическое обоснование: Adam объединяет лучшие черты Momentum и RMSprop. Он хранит экспоненциально затухающее среднее прошлых градиентов (первый момент, m_t), как Momentum, и экспоненциально затухающее среднее квадратов градиентов (второй момент, v_t), как RMSprop. Первый момент помогает ускорять движение в нужном направлении, а второй — адаптировать скорость обучения для каждого параметра.

Формула обновления:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J(\theta_t)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla J(\theta_t))^2$$

С корректировкой смещения (bias correction):

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

где β_1, β_2 — коэффициенты затухания для моментов.

Параметры в коде (PyTorch):

```
optimizer = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-8)
# betas: кортеж коэффициентов (beta1, beta2)
# eps: для численной стабильности

```

9 Источники

Приведённые описания базируются на авторитетных источниках по ML/DL, а практические примеры – на библиотеках scikit-learn, Keras, PyTorch.