Пример: классификация фамилий с помощью MLP

В этом разделе мы воспользуемся MLP для классификации фамилий по стране их происхождения. Вывод демографической информации (такой как национальность) на основе общедоступных данных применяется во множестве сфер, начиная от рекомендаций товаров до обеспечения равных возможностей пользователей из разных стран. Тем не менее демографические характеристики и другие не обезличенные атрибуты относятся к «защищаемым законом атрибутам». Использовать их при моделировании и в программных продуктах следует осторожно[1]. Мы начнем с разбиения фамилий на символы и их обработки методом, аналогичным обработке слов, описанной в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76. Помимо различий в данных, модели на уровне символов практически схожи по структуре и реализации с основанными на словах моделями[2].

Важный урок, который следует вынести из этого примера: реализация и обучение MLP представляет собой очевидное развитие методики реализации и обучения перцептрона из главы 3. На самом деле мы будем на протяжении всей книги вспоминать пример из главы 3 для более подробного описания этих компонентов. Далее, мы не станем включать код, приведенный в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76. Чтобы увидеть код примеров целиком, обратитесь к прилагаемым к книге материалам[3].

Данный раздел начинается с описания набора данных фамилий и его предварительной обработки. Далее мы пошагово опишем конвейер преобразования строки с фамилией в векторизованный мини-пакет с помощью классов `Vocabulary`, `Vectorizer` и `DataLoader`. Прочитавшие главу 3 встретят эти классы как старых приятелей, только с небольшими модификациями.

Далее в этом разделе вас ждет описание модели `surnameclassifier` и обоснование ее архитектуры. MLP здесь схож с примером перцептрона из главы 3, но, помимо изменений модели, в этом примере появятся многоклассовые выходные данные и соответствующие функции потерь. После описания модели мы рассмотрим

---

1 Обсуждение этических вопросов NLP можно найти на сайте ethicsinnlp.org.

2 Интересно, что недавние исследования показали: включение моделей на уровне символов в модели на уровне слов могут повысить эффективность последних. См. книгу Петерса и др. (Peters et al., 2018).

3 См. блокнот /chapters/chapter_4/4_2_mlp_surnames/4_2_Classifying_Surnames_with_an_MLP.ipynb из репозитория GitHub этой книги (https://nlproc.info/PyTorchNLPBook/repo/).

процедуру обучения. Она очень напоминает уже виденную вами в подразделе «Процедура обучения» на с. 114, так что для краткости мы не станем рассматривать ее так подробно, как там. Мы очень рекомендуем вам вернуться к этому подразделу за дополнительными пояснениями.

Завершается этот пример оценкой модели на контрольной порции набора данных и описанием процедуры вывода для новой фамилии. В многоклассовых предсказаниях хорошо то, что можно получить не только наиболее вероятное предсказание, и мы дополнительно обсудим, как вывести k наиболее вероятных предсказаний для новой фамилии.

Набор данных фамилий

В этом примере мы воспользуемся *набором данных фамилий*, состоящим из 10 000 фамилий людей 18 разных национальностей, собранных авторами из различных источников в Интернете. Этот набор данных будет использоваться в нескольких примерах в книге. Он обладает несколькими интересными свойствами. Первое из них — относительная несбалансированность. Три наиболее многочисленных класса охватывают более 60 % данных: 27 % английских, 21 % русских и 14 % арабских. Частота оставшихся 15 национальностей убывает — свойство, присущее и языкам. Второе свойство — достоверная и интуитивно понятная связь между национальностью и написанием фамилии. Многие особенности написания фамилий сильно связаны со страной происхождения (например, O'Neill (О'Нил), Antonopoulos (Антонопулос), Nagasawa (Нагасава) или Zhu (Чжу)).

Начнем создание итогового набора данных с несколько менее обработанной версии, чем та, что предложена в прилагаемых к книге материалах, и произведем несколько операций модификации набора данных. Первая нацелена на исправление дисбаланса — в исходном наборе данных было более 70 % русских фамилий, возможно, из-за систематической ошибки выборки или большого числа неповторяющихся русских фамилий. Для этого выполним прореживание этого слишком широко представленного класса, выбрав случайным образом подмножество фамилий из числа помеченных как русские. Далее группируем набор данных по национальностям и разбиваем его на три фрагмента: 70 % отводим на обучающий набор данных, 15 % на проверочный и 15 % на контрольный, чтобы распределения меток классов между фрагментами были сравнимыми.

Реализация `SurnameDataset` практически идентична классу `ReviewDataset`, показанному в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, с единственными небольшими отличиями в реализации метода `__getitem__()`[1]. Напомним, что представленные в книге классы наборов данных

[1] Изменены также некоторые имена переменных, чтобы отразить их роль/содержимое.

наследуют класс `Dataset` фреймворка PyTorch, поэтому нам нужно реализовать два метода: `__getitem__()`, возвращающий точку данных по индексу, и `__len__()`, возвращающий длину набора данных. Различие этого примера и примера из главы 3 — в методе `__getitem__()` (пример 4.5). Он возвращает не векторизованный обзор, как в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, а векторизованную фамилию и индекс соответствующей национальности.

Пример 4.5. Реализация метода SurnameDataset.__getitem__()

```
class SurnameDataset(Dataset):
    # Реализация практически идентична примеру 3.14

    def __getitem__(self, index):
        row = self._target_df.iloc[index]
        surname_vector = \
            self._vectorizer.vectorize(row.surname)
        nationality_index = \
            self._vectorizer.nationality_vocab.lookup_token(row.nationality)

        return {'x_surname': surname_vector,
                'y_nationality': nationality_index}
```

Классы Vocabulary, Vectorizer и DataLoader

Для классификации фамилий по символам мы воспользуемся классами `Vocabulary`, `Vectorizer` и `DataLoader`, чтобы преобразовать фамилии в виде строковых значений в векторизованные мини-пакеты. Мы используем те же структуры данных, что и в разделе «Пример: классификация тональностей обзоров ресторанов» на с. 76, демонстрируя тем самым полиморфизм, позволяющий обрабатывать символьные токены фамилий аналогично токенам-словам из обзоров Yelp. Вместо векторизации путем отображения токенов-слов в целочисленные значения данные векторизуются с помощью отображения символов в целочисленные значения.

Класс Vocabulary

Используемый в этом примере класс `Vocabulary` в точности аналогичен тому, который использовался в примере 3.16 для отображения слов обзоров Yelp в соответствующие целочисленные значения. Отметим, что `Vocabulary` представляет собой сочетание двух словарей Python, формирующих взаимно однозначное соответствие между токенами (в данном случае символьными) и целочисленными значениями; а именно, первый словарь задает соответствие символов целочисленным индексам, а второй задает соответствие целочисленных индексов символам. Метод `add_token()` служит для добавления новых токенов в `Vocabulary`, метод `lookup_token()` — для извлечения индекса, а `lookup_index()` — для извлечения токена по заданному индексу (что удобно в фазе вывода). В отличие от `Vocabulary`

для обзоров Yelp, мы используем унитарное представление[1], не подсчитываем частоту вхождения символов и ограничиваемся только часто встречающимися элементами. В основном потому, что набор данных невелик и большинство символов встречаются достаточно часто.

Класс SurnameVectorizer

Если класс `Vocabulary` преобразует отдельные токены (символьные) в целые числа, то класс `SurnameVectorizer` отвечает за применение `Vocabulary` и преобразование фамилии в вектор. Создание экземпляра и его использование аналогичны работе класса `ReviewVectorizer` в пункте «Класс Vectorizer» на с. 85, но с одним ключевым отличием: строковое значение не разбивается по пробелам. Фамилии представляют собой последовательности символов, каждый из которых — отдельный токен в нашем словаре. Однако вплоть до раздела «Сверточные нейронные сети» на с. 120 мы будем игнорировать информацию о последовательности и создавать свернутое унитарное представление входных данных, проходя в цикле по всем символам входной строки. Для ранее не встреченных символов мы выделим специальный токен `UNK`. Он используется в словаре символов, поскольку мы создаем экземпляр `Vocabulary` на основе только обучающих данных, и в проверочных/контрольных данных вполне могут оказаться уникальные символы[2].

Следует заметить, что, хотя в этом примере используется свернутое унитарное представление, в последующих главах вы узнаете и о других методах векторизации, альтернативных, а иногда и превосходящих унитарное кодирование. А именно, в разделе «Пример: классификация фамилий с помощью CNN» на с. 130 вы встретите матрицу из уникальных векторов, в которой каждый символ занимает определенную позицию и обладает своим собственным унитарным вектором. Далее, в главе 5, вы узнаете о слое вложений — векторизации, которая возвращает вектор целочисленных значений, и создании на их основе матрицы плотных векторов. А пока рассмотрим код класса `SurnameVectorizer` (пример 4.6).

Пример 4.6. Реализация класса SurnameVectorizer

```
class SurnameVectorizer(object):
    """ Векторизатор, приводящий словари в соответствие друг другу
        и использующий их"""
    def __init__(self, surname_vocab, nationality_vocab):
        self.surname_vocab = surname_vocab
        self.nationality_vocab = nationality_vocab

    def vectorize(self, surname):
```

[1] См. описание унитарных представлений в подразделе «Унитарное представление» на с. 24.

[2] А при нашем разбиении данных в проверочном наборе действительно есть уникальные символы, которые могут привести к сбою обучения, если не использовать UNK.

```
        """ Векторизация передаваемой фамилии

        Аргументы:
            surname (str): фамилия
        Возвращает:
            one_hot (np.ndarray): свернутое унитарное представление
        """
        vocab = self.surname_vocab
        one_hot = np.zeros(len(vocab), dtype=np.float32)
        for token in surname:
            one_hot[vocab.lookup_token(token)] = 1
        return one_hot

    @classmethod
    def from_dataframe(cls, surname_df):
        """ Создает экземпляр векторизатора на основе объекта DataFrame
            набора данных

        Аргументы:
            surname_df (pandas.DataFrame): набор данных фамилий
        Возвращает:
            экземпляр SurnameVectorizer
        """
        surname_vocab = Vocabulary(unk_token="@")
        nationality_vocab = Vocabulary(add_unk=False)

        for index, row in surname_df.iterrows():
            for letter in row.surname:
                surname_vocab.add_token(letter)
            nationality_vocab.add_token(row.nationality)

        return cls(surname_vocab, nationality_vocab)
```

Модель SurnameClassifier

Класс `SurnameClassifier` (пример 4.7) — реализация MLP, представленного ранее в главе. Первый линейный слой отображает входные векторы в промежуточный вектор, к которому применяется нелинейность. Второй линейный слой отображает промежуточный вектор в вектор предсказаний.

На последнем шаге может применяться многомерная логистическая функция для приведения суммы выходных значений к 1, то есть их интерпретации как «вероятностей»[1]. Причина необязательности ее применения кроется в математической формулировке используемой функции потерь — функции потерь на основе

[1] Мы намеренно написали слово «вероятностей» в кавычках, чтобы подчеркнуть, что это вовсе не настоящие вероятности в байесовском смысле, но поскольку сумма выходных значений равна 1, то это допустимое распределение, которое можно интерпретировать как вероятности. Это одно из самых занудных примечаний в данной книге, так что можете смело его игнорировать, просто закройте на это глаза и называйте их вероятностями.

перекрестной энтропии, с которой мы познакомились в разделе «Функции потерь» на с. 64. Напомним, что функция потерь на основе перекрестной энтропии лучше всего подходит для многоклассовой классификации, но вычисление многомерной логистической функции во время обучения — напрасная трата ресурсов. Кроме того, во многих случаях она может приводить к численной неустойчивости.

Пример 4.7. Реализация класса SurnameClassifier с помощью MLP

```
import torch.nn as nn
import torch.nn.functional as F

class SurnameClassifier(nn.Module):
    """ Многослойный перцептрон с двумя слоями для классификации фамилий """
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Аргументы:
            input_dim (int): размер входных векторов
            hidden_dim (int): размер выходных векторов первого линейного слоя
            output_dim (int): размер выходных векторов второго линейного слоя
        """
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход классификатора

        Аргументы:
            x_in (torch.Tensor): входной тензор данных
                Значение x_in.shape должно быть (batch, input_dim)
            apply_softmax (bool): флаг для многомерной логистической функции
            активации. При использовании функции потерь на основе
            перекрестной энтропии должен равняться false
        Возвращает:
            итоговый тензор. Значение tensor.shape должно
            быть (batch, output_dim).
        """
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)

        if apply_softmax:
            prediction_vector = F.softmax(prediction_vector, dim=1)

        return prediction_vector
```

Процедура обучения

Хотя в этом примере мы используем другую модель, набор данных и функцию потерь, процедура обучения не отличается от описанной в предыдущей главе. Поэтому в примере 4.8 мы приведем только `args` и основные различия процедуры обучения между этим примером и примером из раздела «Пример: классификация тональностей обзоров ресторанов» на с. 76.

Пример 4.8. Гиперпараметры и настройки программы для классификатора фамилий на основе MLP

```
args = Namespace(
    # Информация о данных и путях
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch4/surname_mlp",
    # Гиперпараметры модели
    hidden_dim=300
    # Гиперпараметры обучения
    seed=1337,
    num_epochs=100,
    early_stopping_criteria=5,
    learning_rate=0.001,
    batch_size=64,
    # Настройки времени выполнения не приводятся для экономии места
)
```

Наиболее заметное различие в обучении относится к выходным данным модели и используемой функции потерь. В этом примере выходные данные представляют собой вектор многоклассовых предсказаний, который можно преобразовать в вероятности. Для таких выходных данных подходят только функции потерь `CrossEntropyLoss()` и `NLLLoss()`. Воспользуемся первой.

В примере 4.9 приведено создание экземпляров набора данных, модели, функции потерь и оптимизатора. Они практически идентичны тем, что описывались в примере из главы 3. Собственно, это справедливо практически для всех примеров в последующих главах.

Пример 4.9. Создание экземпляров набора данных, модели, функции потерь и оптимизатора

```
dataset = SurnameDataset.load_dataset_and_make_vectorizer(args.surname_csv)
vectorizer = dataset.get_vectorizer()

classifier = SurnameClassifier(input_dim=len(vectorizer.surname_vocab),
                               hidden_dim=args.hidden_dim,
                               output_dim=len(vectorizer.nationality_vocab))

classifier = classifier.to(args.device)

loss_func = nn.CrossEntropyLoss(dataset.class_weights)
optimizer = optim.Adam(classifier.parameters(), lr=args.learning_rate)
```

Цикл обучения

Цикл обучения для этого примера практически идентичен описанному в пункте «Цикл обучения» на с. 92, за исключением названий переменных. А именно, из примера 4.10 видно, что для получения данных из `batch_dict` используются другие ключи. Помимо этого незначительного различия, функциональность цикла

обучения осталась такой же. На основе обучающих данных вычисляются выходные данные модели, потерь и градиентов. Затем модель обновляется на основе градиентов.

Пример 4.10. Фрагмент цикла обучения

```
# Процедура обучения состоит из следующих пяти шагов:

# -------------------------------------
# Шаг 1. Обнуляем градиенты
optimizer.zero_grad()

# Шаг 2. Вычисляем выходные значения
y_pred = classifier(batch_dict['x_surname'])

# Шаг 3. Вычисляем функцию потерь
loss = loss_func(y_pred, batch_dict['y_nationality'])
loss_batch = loss.to("cpu").item()
running_loss += (loss_batch - running_loss) / (batch_index + 1)

# Шаг 4. Получаем градиенты на основе функции потерь
loss.backward()

# Шаг 5. Оптимизатор обновляет значения параметров по градиентам
optimizer.step()
```

Оценка модели и получение предсказаний

Чтобы выяснить эффективность модели, необходимо проанализировать модель с помощью количественных и качественных методов. Количественные методы включают оценку погрешности на специально выделенных контрольных данных для определения того, способен ли классификатор выполнить обобщение на еще не виденные выборки. Качественные методы включают ваше личное представление о том, чему обучилась модель, на основе первых k предсказаний классификатора для новой выборки.

Оценка на контрольном наборе данных

Для оценки `SurnameClassifier` на контрольных данных мы поступим так же, как и в примере с классификацией текстов обзоров ресторанов в подразделе «Оценка, вывод и просмотр» на с. 95: выберем фрагмент данных `'test'`, вызовем метод `classifier.eval()` и пройдем в цикле по контрольным данным аналогично прочим фрагментам. В этом примере фреймворк PyTorch из-за вызова метода `classifier.eval()` не обновляет параметры модели при использовании контрольных/проверочных данных.

Точность модели на контрольных данных составляет около 50 %. Если запустить процедуру обучения из прилагаемого к книге блокнота, вы увидите, что эффективность на обучающих данных выше. Дело в том, что модель всегда лучше приспособлена к данным, на которых обучается, так что эффективность на обучающих данных не показательна для обучения на новых данных. Если вы проверяете работу кода примеров, рекомендуем попробовать различные размеры скрытого измерения. При этом вы должны наблюдать рост эффективности[1]. Впрочем, этот рост не очень велик (особенно по сравнению с моделью из раздела «Пример: классификация фамилий с помощью CNN» на с. 130). Основная причина: неудачность свернутой унитарной векторизации как метода представления. Хотя каждая из фамилий компактно представляется в виде одного вектора, при этом теряется информация об упорядоченности символов, играющая большое значение при идентификации происхождения фамилии.

Классификация новой фамилии

Пример 4.11 демонстрирует код для классификации новой фамилии. Функция сначала запускает процесс векторизации по отношению к полученной в виде строкового значения фамилии, а затем получает предсказания модели. Обратите внимание на включенный флаг `apply_softmax` — это означает, что переменная `result` содержит вероятности. Предсказание модели в полиномиальном случае представляет собой список вероятностей различных классов. Для получения оптимального класса, соответствующего наибольшей предсказанной вероятности, воспользуемся функцией `max()` тензора PyTorch.

Пример 4.11. Вывод с помощью существующей модели (классификатора). Предсказание национальности по фамилии

```
def predict_nationality(name, classifier, vectorizer):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    result = classifier(vectorized_name, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality,
            'probability': probability_value}
```

[1] Напомним уже сказанное в главе 3: при экспериментах с гиперпараметрами, такими как размер скрытого измерения и количество слоев, оценка должна производиться на проверочном наборе данных, а не на контрольном. Если же текущие гиперпараметры вас устраивают, можете выполнить оценку на контрольных данных.

Извлечение k наилучших предсказаний для новой фамилии

Часто бывает полезно взглянуть не только на одно лучшее предсказание. Например, в NLP стандартной практикой считается извлечение k наилучших предсказаний и повторное ранжирование их с помощью другой модели. Для получения этих предсказаний во фреймворке PyTorch есть удобная функция `torch.topk()` (пример 4.12).

Пример 4.12. Предсказание k лучше всего подходящих национальностей

```
def predict_topk_nationality(name, classifier, vectorizer, k=5):
    vectorized_name = vectorizer.vectorize(name)
    vectorized_name = torch.tensor(vectorized_name).view(1, -1)
    prediction_vector = classifier(vectorized_name, apply_softmax=True)
    probability_values, indices = torch.topk(prediction_vector, k=k)

    # Возвращаемый размер 1,k
    probability_values = probability_values.detach().numpy()[0]
    indices = indices.detach().numpy()[0]

    results = []
    for prob_value, index in zip(probability_values, indices):
        nationality = vectorizer.nationality_vocab.lookup_index(index)
        results.append({'nationality': nationality,
                        'probability': prob_value})

    return results
```

Регуляризация многослойных перцептронов: регуляризация весов и структурная регуляризация

В главе 3 мы объяснили, почему регуляризация является решением проблемы переобучения, и изучили два важных типа регуляризации весов — L1 и L2. Эти методы регуляризации весов применимы как к MLP, так и к сверточным нейронным сетям, которые мы рассмотрим в следующем разделе. Помимо регуляризации весов, для глубоких моделей (то есть моделей из нескольких слоев), таких как обсуждавшиеся в этой главе упреждающие сети, важнейшее значение имеет подход структурной регуляризации под названием *дропаут* (dropout).

Попросту говоря, во время обучения дропаут отбрасывает на вероятностной основе связи между элементами из двух смежных слоев. В чем польза такого подхода? Начнем с интуитивно понятного (юмористического) объяснения Стивена Мерити (Stephen Merity)[1]: *«Дропаут, попросту говоря, означает, что если у вас хорошо*

[1] Это определение взято из весьма забавной первоапрельской «статьи» Стивена Мерити (http://bit.ly/2Cq1FJR).

получается выполнять какую-либо задачу, будучи мертвецки пьяным, то в трезвом состоянии вы тем более сможете ее выполнить. Этому наблюдению обязаны собой множество достижений современной науки, так что зарождается целое движение против использования дропаута в нейронных сетях».

Нейронные сети — особенно глубокие сети с большим количеством слоев — способны создавать интересные коадаптации элементов. «Коадаптация» — термин из нейронаук, означающий просто ситуацию, при которой связь между двумя элементами становится чрезвычайно сильной за счет ослабления связей между другими элементами. Обычно это приводит к переобучению модели на текущих данных. Вероятностное отбрасывание связей между элементами позволяет гарантировать, что никакой конкретный элемент не окажется в постоянной зависимости от другого конкретного элемента, благодаря чему модели становятся ошибкоустойчивыми. Дропаут не требует добавления в модель дополнительных параметров, а лишь одного гиперпараметра — «вероятности отброса»[1]. Это, как легко догадаться, вероятность, при которой отбрасываются связи между элементами. Обычно ее задают равной `0.5`. Пример 4.13 демонстрирует реализацию MLP с дропаутом.

Пример 4.13. MLP с дропаутом

```
import torch.nn as nn
import torch.nn.functional as F

class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        Аргументы:
            input_dim (int): размер входных векторов
            hidden_dim (int): размер на выходе первого линейного слоя
            output_dim (int): размер на выходе второго линейного слоя
        """
        super(MultilayerPerceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """ Прямой проход MLP

        Аргументы:
            x_in (torch.Tensor): входной тензор данных
                Значение x_in.shape должно быть (batch, input_dim)
            apply_softmax (bool): флаг для многомерной логистической функции
            активации. При использовании функции потерь на основе
            перекрестной энтропии должен равняться false
        Возвращает:
```

[1] Некоторые библиотеки глубокого обучения по непонятной причине называют (и интерпретируют) эту вероятность «вероятностью сохранения», то есть придают ей в точности противоположный смысл.

```
        Итоговый тензор. Значение tensor.shape должно
        быть (batch, output_dim).
    """
    intermediate = F.relu(self.fc1(x_in))
    output = self.fc2(F.dropout(intermediate, p=0.5))

    if apply_softmax:
        output = F.softmax(output, dim=1)
    return output
```

Важно отметить, что *дропаут применяется только во время обучения, а не оценки*. В качестве упражнения рекомендуем вам поэкспериментировать с моделью `SurnameClassifier` с дропаутом и посмотреть, как изменятся результаты.

Сверточные нейронные сети

В первой части главы мы подробно изучили MLP — нейронные сети, построенные из ряда линейных слоев и нелинейных функций. MLP — не лучший инструмент для того, чтобы использовать возможности последовательных паттернов[1]. Например, в наборе данных фамилий определенные участки могут в значительной степени раскрывать национальность их носителей (как O' в O'Neil, opulos в Antonopulos, sawa в Nagasawa или Zh в Zhu). Длины этих участков могут быть различными, сложность состоит в том, чтобы уловить их без явного кодирования.

Рассмотрим сверточные нейронные сети — тип нейронных сетей, хорошо подходящий для обнаружения пространственной субструктуры (а значит, и создания осмысленной пространственной субструктуры). CNN добиваются этого за счет небольшого количества весов, используемых для просмотра входящих тензоров данных. В результате этого просмотра генерируются выходные тензоры, отражающие обнаружение (или не обнаружение) субструктур.

Мы начнем с описания способов функционирования CNN и вопросов их проектирования. Подробно обсудим гиперпараметры CNN, чтобы вы научились интуитивно чувствовать их поведение и влияние на выходные результаты. И наконец, мы шаг за шагом пройдем по нескольким простым примерам, иллюстрирующим механизм работы CNN. В разделе «Пример: классификация фамилий с помощью CNN» на с. 130 мы рассмотрим более масштабный пример.

---

[1] Можно спроектировать такой MLP, который бы принимал на входе символьные биграммы, чтобы уловить некоторые из подобных зависимостей. Количество биграмм для английского языка (алфавит которого состоит из 26 букв) составляет 325. Так что при скрытом слое из 100 узлов число параметров для входного скрытого слоя будет 325×100. А если рассматривать все возможные символьные триграммы, то получим еще 2600×100 параметров. Как мы увидим, сверточные нейронные сети позволяют уловить ту же информацию при намного меньшем числе параметров благодаря совместному использованию параметров.