1. **Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.**

**Модель перцептрона** – это простая форма искусственной нейронной сети (ИНС), которая используется для решения задач классификации. Он состоит из одного или нескольких искусственных нейронов, называемых перцептронами, которые принимают входные данные, вычисляют взвешенную сумму этих данных и применяют функцию активации для определения выходного значения.

Проблема линейно неразделимых множеств возникает, когда данные невозможно разделить линейной границей на два класса. Примером такой проблемы может быть задача классификации изображений, где объекты разных классов перекрываются или имеют сложную геометрию. Простой перцептрон не способен решить такую задачу, поскольку он может только создавать линейные разделяющие поверхности.

Однако, данную проблему можно решить с помощью многослойной нейронной сети (Многослойной ИНС), такой как перцептрон с прямой связью (feedforward neural network). Многослойная ИНС состоит из нескольких слоев нейронов, включая входной слой, скрытые слои и выходной слой. Каждый нейрон в слое связан с нейронами предыдущего слоя, и информация передается от входного слоя к выходному слою через промежуточные слои. В многослойной ИНС проблема линейно неразделимых множеств решается путем добавления нелинейности в модель. Это достигается использованием функций активации, которые вводят нелинейные преобразования в выходные значения нейронов. Такие функции активации, как сигмоидная функция или функция ReLU (Rectified Linear Unit), позволяют ИНС моделировать сложные нелинейные зависимости между данными и принимать решения на основе этой информации.

Пример кода, демонстрирующий создание и обучение многослойной нейронной сети:

# Загрузка данных

train\_dataset = …

test\_dataset = …

train\_dataloader = …

test\_dataloader = …

# Определение модели

class NeuralNet(nn.Module):

def \_\_init\_\_(self):

super(NeuralNet, self).\_\_init\_\_()

self.fc1 = nn.Linear(784, 256)

self.fc2 = nn.Linear(256, 128)

self.fc3 = nn.Linear(128, 10)

def forward(self, x):

x = x.view(x.size(0), -1)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

model = NeuralNet()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Обучение модели

num\_epochs = 10

device=torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model.to(device)

for epoch in range(num\_epochs):

running\_loss = 0.0

for images, labels in train\_dataloader:

images = images.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {running\_loss/len(train\_dataloader)}')

# Тестирование модели

model.eval()

with torch.no\_grad():

correct = 0

total = 0

for images, labels in test\_dataloader:

images = images.to(device)

labels = labels.to(device)

outputs = model(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Accuracy: {accuracy}%')

Этот код демонстрирует создание простой многослойной нейронной сети с тремя полносвязными слоями. Она обучается на наборе данных MNIST для классификации рукописных цифр. Каждая эпоха обучения выводит среднюю потерю и, по окончании обучения, показывает точность модели на тестовом наборе данных.

1. **Линейное отображение. Векторно-матричное дифференцирование.**

Линейное отображение является основным понятием в линейной алгебре и математическом анализе. В контексте машинного обучения и глубокого обучения, линейное отображение часто используется в нейронных сетях для выполнения линейных преобразований данных.

Линейное отображение представляет собой математическую функцию, которая преобразует входные данные в соответствии с линейной формулой. Векторно-матричное дифференцирование относится к процессу нахождения производных от выражений, содержащих матрицы и векторы. Оно является важной частью обучения нейронных сетей, так как позволяет вычислять градиенты и обновлять веса модели во время обучения.

Вот пример кода, использующий библиотеку PyTorch, для демонстрации линейного отображения и векторно-матричного дифференцирования:

# Определение линейного отображения

linear = nn.Linear(10, 5)

# Создание случайного входного тензора размером (batch\_size, input\_size)

input\_tensor = torch.randn(32, 10)

# Применение линейного отображения к входному тензору

output\_tensor = linear(input\_tensor)

# Вывод размерности выходного тензора

print("Output Tensor Size:", output\_tensor.size())

# Расчет градиентов по весам и входным данным

output\_tensor.mean().backward()

# Вывод градиентов по весам

print("Gradients w.r.t. Weights:", linear.weight.grad)

# Вывод градиентов по входным данным

print("Gradients w.r.t. Input:", input\_tensor.grad)

В этом примере мы используем библиотеку PyTorch для создания линейного отображения с помощью класса nn.Linear. Затем мы создаем случайный входной тензор размером (32, 10) и применяем линейное отображение к этому входному тензору. Мы выводим размерность выходного тензора и затем выполняем обратное распространение градиентов для вычисления градиентов по весам и входным данным. Результаты градиентов выводятся на экран.

1. **Функции активации. Требования к функциям активации Популярные функции активации.**

**Функции активации** – это математические функции, которые применяются к выходным значениям нейронов в искусственных нейронных сетях (ИНС). Они определяют, какой будет выход нейрона в зависимости от его входа.

Требования к функциям активации:

*1. Нелинейность:* Функции активации должны быть нелинейными, чтобы ИНС могла моделировать сложные нелинейные зависимости в данных. Если бы функции активации были линейными, вся сеть свелась бы к линейной модели, что ограничило бы ее способность решать сложные задачи.

*2. Непрерывность и дифференцируемость:* Для эффективного обучения с использованием алгоритмов градиентного спуска функции активации должны быть непрерывными и дифференцируемыми. Это позволяет вычислять градиенты и обновлять веса сети в процессе обучения.

Несколько популярных функций активации в нейронных сетях:

*1. Сигмоидная функция (Sigmoid):*

- Формула: f(x) = 1 / (1 + exp(-x))

- Описание: Сигмоидная функция преобразует входные значения в интервал от 0 до 1. Она используется преимущественно в задачах бинарной классификации, где требуется вероятностная интерпретация выхода.

import torch

import torch.nn as nn

model = nn.Sequential(

nn.Linear(64, 64),

nn.Sigmoid(), ...)

*2. Гиперболический тангенс (Tanh):*

- Формула: f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))

- Описание: Гиперболический тангенс имеет форму сигмоидной кривой, но выходит в интервал от -1 до 1. Он обладает центрированным нулем, что полезно при моделировании данных со смещением.

import torch

import torch.nn as nn

model = nn.Sequential(

nn.Linear(64, 64),

nn.Tanh(), ...)

*3. Функция ReLU (Rectified Linear Unit):*

- Формула: f(x) = max(0, x)

- Описание: Функция ReLU возвращает входное значение, если оно положительное, и 0 в противном случае. Она является простой и вычислительно эффективной функцией активации и широко используется в глубоком обучении.

import torch

import torch.nn as nn

model = nn.Sequential(

nn.Linear(64, 64),

nn.ReLU(), ...)

*4. Softmax:*

- Формула: f(x) = exp(x) / sum(exp(x))

- Описание: Softmax используется для многоклассовой классификации. Она преобразует входные значения в вероятности, сумма которых равна 1. Softmax часто применяется на выходном слое нейронной сети.

import torch

import torch.nn as nn

model = nn.Sequential(

nn.Linear(64, 10),

nn.Softmax(dim=1), ...)

Это лишь некоторые из множества доступных функций активации. Выбор функции активации зависит от конкретной задачи, типа данных и особенностей модели, и может потребовать экспериментирования для определения наилучшего варианта.

1. **Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.**

Адаптивные методы градиентного спуска являются улучшениями стохастического градиентного спуска (SGD) и направлены на более эффективную оптимизацию моделей машинного обучения. Два популярных адаптивных метода градиентного спуска — это **метод импульсов (Momentum)** и **метод Нестерова (Nesterov Accelerated Gradient)**.

*1. Метод импульсов (Momentum):* Метод импульсов добавляет "импульс" к обычному градиентному спуску для ускорения и стабилизации обучения. Он учитывает предыдущие изменения весов и использует их для обновления текущих градиентов. Это позволяет преодолеть проблемы с большими значениями градиента и осцилляциями при движении по склону функции потерь. Метод импульсов выражается в следующей формуле обновления весов:

v = beta \* v - learning\_rate \* gradient

weights += v

где `v` - импульс, `beta` - коэффициент импульса (обычно выбирается в диапазоне от 0.9 до 0.99), `learning\_rate` - скорость обучения и `gradient` - градиент функции потерь.

*2. Метод Нестерова (Nesterov Accelerated Gradient):* Метод Нестерова является улучшением метода импульсов. Он предсказывает будущее положение весов, основываясь на их текущем положении и направлении изменения. Затем вычисляет градиент в этом "предсказанном" положении и использует его для обновления весов. Это позволяет более точно и быстрее двигаться в направлении оптимума функции потерь. Формула обновления весов для метода Нестерова выглядит следующим образом:

v = beta \* v - learning\_rate \* gradient(weights + beta \* v)

weights += v

где `v` - импульс, `beta` - коэффициент импульса, `learning\_rate` - скорость обучения и `gradient(weights + beta \* v)` - градиент функции потерь в "предсказанном" положении.

Пример кода с использованием библиотеки PyTorch, демонстрирующий методы импульсов и Нестерова:

# Создание модели

model = nn.Linear(10, 1)

# Определение функции потерь

criterion = nn.MSELoss()

# Определение оптимизатора с методом импульсов

optimizer\_momentum = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Определение оптимизатора с методом Нестерова

optimizer\_nesterov = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, nesterov=True)

# Обучение модели с методом импульсов

for epoch in range(num\_epochs):

optimizer\_momentum.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, targets)

loss.backward()

optimizer\_momentum.step()

# Обучение модели с методом Нестерова

for epoch in range(num\_epochs):

optimizer\_nesterov.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, targets)

loss.backward()

optimizer\_nesterov.step()

В этом примере мы создаем модель с помощью `nn.Linear`, определяем функцию потерь `nn.MSELoss` и два оптимизатора: один с методом импульсов (`optim.SGD`) и другой с методом Нестерова (`optim.SGD` с параметром `nesterov=True`). Затем мы выполняем обучение модели с использованием соответствующих оптимизаторов.

1. **Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.**

Глубокое обучение является разделом машинного обучения, который занимается обучением и использованием глубоких нейронных сетей. Глубокие нейронные сети состоят из множества слоев, которые выполняют последовательную обработку входных данных для извлечения высокоуровневых признаков и принятия решений. Глубокое обучение обрело огромную популярность в последние годы, и это можно назвать "второй весной искусственного интеллекта". Несколько причин, которые способствовали этому росту, включают:

*- Крупномасштабные данные:* В настоящее время доступно большое количество данных, и глубокое обучение может извлекать полезную информацию из этих данных для принятия более точных и сложных решений.

*- Вычислительная мощность:* Современные графические процессоры (GPU) и специализированные аппаратные ускорители (например, TPU) предоставляют возможность эффективного обучения и применения глубоких нейронных сетей.

*- Архитектурные инновации:* Были разработаны новые архитектуры нейронных сетей, такие как сверточные нейронные сети (CNN), рекуррентные нейронные сети (RNN) и глубокие сверточные генеративные модели (DCGAN), которые демонстрируют высокую эффективность в различных задачах обработки данных.

*- Прогресс в исследованиях:* Множество исследований в области глубокого обучения привело к разработке новых методов и техник, улучшающих производительность и эффективность глубоких нейронных сетей.

# Определение модели

class NeuralNet(nn.Module):

def \_\_init\_\_(self):

super(NeuralNet, self).\_\_init\_\_()

self.flatten = nn.Flatten()

self.fc1 = nn.Linear(784, 128)

self.relu = nn.ReLU()

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

x = self.flatten(x)

x = self.fc1(x)

x = self.relu(x)

x = self.fc2(x)

return x

model = NeuralNet()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Обучение модели

num\_epochs = 10

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model.to(device)

for epoch in range(num\_epochs):

running\_loss = 0.0

for images, labels in train\_dataloader:

images = images.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

print(f'Epoch [{epoch + 1}/{num\_epochs}], Loss: {running\_loss / len(train\_dataloader)}')

# Тестирование модели

model.eval()

with torch.no\_grad():

correct = 0

total = 0

for images, labels in test\_dataloader:

images = images.to(device)

labels = labels.to(device)

outputs = model(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Accuracy: {accuracy}%')

# Вывод структуры модели

summary(model, input\_size=(1, 28, 28))

В этом примере мы используем библиотеку PyTorch для создания простой глубокой нейронной сети с двумя полносвязными слоями. Мы обучаем модель на наборе данных MNIST для классификации рукописных цифр. Код выполняет компиляцию модели с оптимизатором Adam и функцией потерь CrossEntropyLoss. Затем модель обучается на тренировочных данных и оценивается на тестовых данных, выводя точность модели на тестовом наборе данных. Также выводится структура модели с помощью функции summary().

1. **Преобразование Softmax и функция потерь Cross Entropy loss.**

Преобразование Softmax (Softmax Transformation) и функция потерь Cross Entropy loss (Перекрёстная энтропия) являются важными компонентами в задачах классификации многих классов в глубоком обучении. Преобразование Softmax применяется к выходам модели для преобразования их в вероятности, где каждое значение представляет вероятность принадлежности объекта к определенному классу. Преобразование Softmax определяется следующей формулой: $$\text{Softmax}(x\_i)=\frac{e^{x\_i}}{\sum\_{j=1}^{N} e^{x\_j}},$$

где $x\_i$ - значение выхода модели для класса $i$, а $N$ - общее количество классов. Преобразование Softmax обеспечивает неотрицательные и нормализованные вероятности для каждого класса, сумма которых равна 1.

Функция потерь Cross Entropy loss используется для оценки расхождения между предсказанными вероятностями и истинными метками классов. Для задачи классификации многих классов функция потерь Cross Entropy loss определяется следующей формулой: $$\text{Loss} = -\sum\_{i=1}^{N} y\_i \log(p\_i),$$

где $y\_i$ - истинная метка класса для класса $i$, а $p\_i$ - предсказанная вероятность класса $i$. Функция потерь Cross Entropy loss стремится минимизировать расхождение между предсказанной вероятностью и истинной меткой класса.

Вот пример кода, использующий библиотеку PyTorch, для демонстрации преобразования Softmax и функции потерь Cross Entropy loss:

# Задание выходов модели

outputs = torch.tensor([[3.0, 1.0, 0.2], [1.5, 2.0, 0.5]])

# Применение преобразования Softmax

softmax = nn.Softmax(dim=1)

probabilities = softmax(outputs)

# Вывод преобразованных вероятностей

print("Probabilities:", probabilities)

# Задание истинных меток классов

labels = torch.tensor([0, 1])

# Вычисление функции потерь Cross Entropy loss

criterion = nn.CrossEntropyLoss()

loss = criterion(outputs, labels)

# Вывод функции потерь

print("Loss:", loss.item())

В этом примере мы задаем выходы модели `outputs`, которые представляют собой необработанные значения перед применением преобразования Softmax. Затем мы используем модуль `nn.Softmax` для применения преобразования Softmax и получения вероятностей `probabilities`. Далее мы задаем истинные метки классов `labels` и определяем функцию потерь `nn.CrossEntropyLoss()`. Мы вычисляем функцию потерь `loss` путем передачи выходов модели и истинных меток классов в функцию потерь. Выводим преобразованные вероятности `probabilities` и функцию потерь `loss`.

1. **Механизм обратного распространения ошибки.**

Механизм обратного распространения ошибки (Backpropagation) является ключевым алгоритмом в обучении нейронных сетей. Он позволяет эффективно вычислять градиенты функции потерь по весам модели, что позволяет обновлять веса сети в направлении уменьшения потерь и улучшения ее производительности. Процесс обратного распространения ошибки состоит из нескольких шагов:

*1. Прямой проход (Forward Pass):* Для заданного входного примера пропускаем его через нейронную сеть, применяя активационные функции и собирая выходные значения каждого слоя.

*2. Вычисление потери (Loss Computation):* Сравниваем предсказанные значения с истинными метками, используя функцию потерь, чтобы определить, насколько сильно модель ошибается.

*3. Обратное распространение (Backward Pass):* Вычисляем градиенты функции потерь по весам модели с помощью правила цепочки (Chain Rule). Градиенты передаются назад через сеть, вычисляя градиенты для каждого слоя.

*4. Обновление весов (Weight Update):* Используя градиенты, вычисленные на предыдущем шаге, обновляем веса модели с помощью оптимизационного алгоритма, например, стохастического градиентного спуска (SGD) или его вариантов.

Пример кода с использованием библиотеки PyTorch, демонстрирующий механизм обратного распространения ошибки:

# Задание входных данных и истинных меток

inputs = torch.tensor([[1.0, 2.0, 3.0]])

labels = torch.tensor([[0.5]])

# Определение модели

model = nn.Linear(3, 1)

criterion = nn.MSELoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

# Процесс обучения

for epoch in range(100):

# Прямой проход

outputs = model(inputs)

# Вычисление потери

loss = criterion(outputs, labels)

# Обратное распространение

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Получение предсказанных значений

predicted = model(inputs)

# Вывод результатов

print("Predicted:", predicted)

В этом примере мы задаем входные данные `inputs` и истинные метки `labels`. Определяем модель, функцию потерь и оптимизатор. Затем мы запускаем цикл обучения, в котором происходит прямой проход, вычисление потери, обратное распространение и обновление весов с помощью оптимизатора. После завершения обучения мы используем модель для получения предсказанных значений `predicted` и выводим их.

1. **Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.**

Кросс-валидация является методом оценки производительности модели машинного обучения на основе доступного набора данных. Она помогает оценить, насколько хорошо модель обобщает данные, которые она не видела ранее. Кросс-валидация обычно включает разделение набора данных на несколько подмножеств (train, validation, test) и проведение обучения и оценки на этих подмножествах.

*1. Обучающая выборка (train):* Это подмножество данных, используемое для обучения модели. Модель настраивает свои параметры на этой выборке путем минимизации функции потерь.

*2. Валидационная выборка (validation):* Это подмножество данных, используемое для настройки гиперпараметров модели и оценки ее производительности в процессе обучения. Она помогает контролировать переобучение и выбирать лучшие параметры модели.

*3. Тестовая выборка (test):* Это независимое подмножество данных, которое не используется в процессе обучения или настройки модели. Она используется для окончательной оценки производительности модели после ее обучения.

Проблема переобучения возникает, когда модель хорошо работает на обучающей выборке, но плохо обобщает на новые данные. Это может привести к низкой производительности модели на реальных задачах. Кросс-валидация и валидационная выборка помогают контролировать переобучение и оценивать производительность модели на независимых данных.

**Ранняя остановка (Early Stopping)** — это техника, которая используется для предотвращения переобучения и нахождения оптимального количества эпох обучения. Она заключается в отслеживании производительности модели на валидационной выборке в процессе обучения. Если производительность на валидационной выборке начинает ухудшаться после некоторого количества эпох, обучение останавливается, чтобы избежать переобучения и сохранить модель с лучшей производительностью.

Вот пример кода для демонстрации кросс-валидации и ранней остановки:

# Загрузка данных и разделение на train, validation, test

data = ...

labels = ...

train\_data, test\_data, train\_labels, test\_labels = train\_test\_split(data, labels, test\_size=0.2, random\_state=42)

train\_data, val\_data, train\_labels, val\_labels = train\_test\_split(train\_data, train\_labels, test\_size=0.2, random\_state=42)

# Определение модели

model = nn.Sequential(

nn.Linear(10, 32),

nn.ReLU(),

nn.Linear(32, 64),

nn.ReLU(),

nn.Linear(64, 1),

nn.Sigmoid())

# Определение функции потерь и оптимизатора

criterion = nn.BCELoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Обучение модели с ранней остановкой

best\_loss = float('inf')

early\_stopping\_patience = 5

epochs\_without\_improvement = 0

for epoch in range(100):

model.train()

optimizer.zero\_grad()

outputs = model(train\_data)

loss = criterion(outputs, train\_labels)

loss.backward()

optimizer.step()

model.eval()

val\_outputs = model(val\_data)

val\_loss = criterion(val\_outputs, val\_labels)

if val\_loss < best\_loss:

best\_loss = val\_loss

epochs\_without\_improvement = 0

else:

epochs\_without\_improvement += 1

if epochs\_without\_improvement == early\_stopping\_patience:

print("Early stopping. No improvement for {} epochs.".format(early\_stopping\_patience))

break

# Оценка производительности на тестовой выборке

model.eval()

test\_outputs = model(test\_data)

test\_loss = criterion(test\_outputs, test\_labels)

print("Test Loss:", test\_loss.item())

В этом примере мы используем модуль `sklearn` для разделения данных на train, validation и test выборки. Затем мы определяем модель с несколькими слоями и выбираем функцию потерь и оптимизатор. В процессе обучения мы вычисляем потери на train и validation выборках и проверяем, улучшается ли производительность модели на validation выборке. Если производительность не улучшается в течение заданного количества эпох, обучение останавливается. Затем мы оцениваем производительность модели на test выборке.

1. **Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.**

Проблема поиска градиента в общей логике обучения нейронной сети заключается в нахождении оптимальных значений весов модели, чтобы минимизировать функцию потерь. Градиент функции многих переменных является важной концепцией для оптимизации параметров модели. Он позволяет вычислить направление и величину наиболее быстрого возрастания функции. Цель состоит в том, чтобы найти направление наискорейшего убывания функции потерь и обновить веса модели в этом направлении.

В PyTorch вычисление градиента функции многих переменных осуществляется автоматически с помощью автоматического дифференцирования (Autograd). PyTorch автоматически отслеживает операции над тензорами и создает вычислительный граф, который позволяет вычислять градиенты.

Вот пример кода, использующий библиотеку PyTorch, для демонстрации вычисления градиента функции многих переменных:

# Определение функции

def func(x):

return x[0] \*\* 2 + 2 \* x[1] \*\* 3

# Создание тензора переменных

x = torch.tensor([2.0, 3.0], requires\_grad=True)

# Вычисление градиента

gradient = torch.autograd.grad(func(x), x)

# Вывод градиента

print("Gradient:", gradient)

В этом примере мы определяем функцию `func(x)`, которая зависит от двух переменных `x[0]` и `x[1]`. Затем мы создаем тензор `x` с указанием `requires\_grad=True`, чтобы указать PyTorch отслеживать градиенты по этим переменным. После этого мы используем функцию `torch.autograd.grad()` для вычисления градиента функции `func` по переменным `x`. Результатом будет градиент функции по каждой переменной.

1. **Дифференцируемое программирование и реализация обратного распространения ошибки.**

**Дифференцируемое программирование (Differentiable Programming)** — это подход, который объединяет идеи из области глубокого обучения и символьной математики для создания моделей, которые могут быть автоматически дифференцированы.

Основной принцип дифференцируемого программирования заключается в том, что все операции и функции, используемые в модели, должны быть дифференцируемыми. Это позволяет использовать алгоритм обратного распространения ошибки для эффективного вычисления градиентов и обновления параметров модели.

PyTorch является одной из популярных библиотек для дифференцируемого программирования. Она предоставляет удобный и гибкий интерфейс для создания и обучения нейронных сетей с помощью автоматического дифференцирования.

Вот пример кода, использующий библиотеку PyTorch для демонстрации дифференцируемого программирования и реализации обратного распространения ошибки:

# Задание входных данные и истинных меток

x = torch.tensor([2.0], requires\_grad=True)

y\_true = torch.tensor([3.0])

# Определение модели

def model(x):

return x \*\* 2

# Прямой проход

y\_pred = model(x)

# Вычисление потери

loss = (y\_pred - y\_true) \*\* 2

# Обратное распространение

loss.backward()

# Получение градиента

gradient = x.grad

# Вывод градиента

print("Gradient:", gradient.item())

В этом примере мы задаем входные данные `x` и истинные метки `y\_true`. Затем мы определяем модель, в данном случае простую квадратичную функцию `model(x) = x \*\* 2`.

Далее мы выполняем прямой проход, вычисляем потерю и применяем обратное распространение с помощью метода `backward()`. В результате вызова `backward()` градиент потери по переменной `x` вычисляется автоматически. Затем мы получаем градиент `x.grad` и выводим его значение.

1. **Стохастический градиентный спуск. Батчи обучающей выборки.**

**Стохастический градиентный спуск (Stochastic Gradient Descent, SGD)** — это один из популярных методов оптимизации в машинном обучении. Он используется для обновления весов модели на основе градиентов функции потерь.

Вместо вычисления градиента по всей обучающей выборке, SGD вычисляет градиенты по отдельным мини-пакетам данных, называемым батчами. Батчи представляют собой случайные подмножества данных из обучающей выборки. Это позволяет SGD обновлять веса модели на каждом шаге обучения и эффективно обрабатывать большие наборы данных.

Пример кода на языке Python с использованием библиотеки PyTorch, демонстрирующий использование SGD и работу с батчами обучающей выборки:

# Загрузка данных и создание DataLoader

X\_train = torch.tensor([[1.0], [2.0], [3.0], [4.0], [5.0]])

y\_train = torch.tensor([[2.0], [4.0], [6.0], [8.0], [10.0]])

dataset = TensorDataset(X\_train, y\_train)

dataloader = DataLoader(dataset, batch\_size=2, shuffle=True)

# Определение модели

model = nn.Linear(1, 1)

criterion = nn.MSELoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

# Процесс обучения

num\_epochs = 10

for epoch in range(num\_epochs):

for batch\_X, batch\_y in dataloader:

# Прямой проход

outputs = model(batch\_X)

# Вычисление потери

loss = criterion(outputs, batch\_y)

# Обратное распространение и обновление весов

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Получение предсказанных значений

X\_test = torch.tensor([[6.0], [7.0], [8.0]])

y\_pred = model(X\_test)

# Вывод результатов

print("Predicted:", y\_pred)

В этом примере мы загружаем данные `X\_train` и `y\_train` и создаем `TensorDataset`, который объединяет входные данные и метки классов в единый датасет. Затем мы создаем `DataLoader` с размером батча 2 и случайной перестановкой данных. Мы определяем модель с одним входным и выходным слоем (`nn.Linear(1, 1)`). Затем мы определяем функцию потерь и оптимизатор SGD. В цикле обучения мы проходим по каждому батчу в `dataloader`, выполняем прямой проход, вычисляем потерю, выполняем обратное распространение и обновляем веса с помощью оптимизатора. После завершения обучения мы используем модель для получения предсказанных значений `y\_pred` на тестовых данных `X\_test` и выводим их. Использование батчей позволяет эффективно обрабатывать большие наборы данных и ускоряет обучение модели.

1. **Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.**

Проблема инициализации весов при обучении искусственных нейронных сетей (ИНС) заключается в выборе начальных значений для весов нейронов. Правильная инициализация весов является важным шагом, поскольку неправильные начальные значения могут затруднить или замедлить процесс обучения.

Когда веса инициализируются слишком малыми значениями, это может привести к проблеме затухания градиента (vanishing gradient problem), когда градиенты сигнала становятся очень малыми по мере его прохождения через слои нейронной сети. В результате обновление весов становится очень медленным, и сеть может не достичь хорошей производительности. С другой стороны, если веса инициализируются слишком большими значениями, это может привести к проблеме взрывного градиента (exploding gradient problem), когда градиенты становятся очень большими и обновление весов становится нестабильным. Это может привести к расходящемуся обучению и нестабильным предсказаниям сети.

Инициализация Ксавье (Xavier initialization) является одним из методов инициализации весов, разработанным для решения проблемы затухания и взрыва градиента. Идея заключается в том, чтобы инициализировать веса таким образом, чтобы дисперсия выходного сигнала каждого нейрона была примерно одинаковой на всех слоях нейронной сети. При использовании инициализации Ксавье веса инициализируются случайными значениями из нормального распределения с нулевым средним и дисперсией, вычисляемой на основе количества входных и выходных связей для каждого нейрона. Формула для вычисления дисперсии весов при инициализации Ксавье зависит от функции активации, используемой в нейронной сети. Применение инициализации Ксавье помогает более стабильно и эффективно обучать нейронные сети, уменьшая проблемы затухания и взрыва градиента. Пример:

import torch

import torch.nn as nn

# Определение нейронной сети

class NeuralNetwork(nn.Module):

def \_\_init\_\_(self):

super(NeuralNetwork, self).\_\_init\_\_()

self.linear1 = nn.Linear(in\_features=784, out\_features=256)

self.linear2 = nn.Linear(in\_features=256, out\_features=128)

self.linear3 = nn.Linear(in\_features=128, out\_features=10)

# Инициализация весов с помощью инициализации Ксавье

nn.init.xavier\_uniform\_(self.linear1.weight)

nn.init.xavier\_uniform\_(self.linear2.weight)

nn.init.xavier\_uniform\_(self.linear3.weight)

def forward(self, x):

x = torch.relu(self.linear1(x))

x = torch.relu(self.linear2(x))

x = self.linear3(x)

return x

# Создание экземпляра нейронной сети

model = NeuralNetwork()

# Вывод инициализированных весов

print(model.linear1.weight)

print(model.linear2.weight)

print(model.linear3.weight)

В этом примере мы создаем класс NeuralNetwork, который наследуется от nn.Module в PyTorch. В конструкторе класса определяются слои нейронной сети (linear1, linear2, linear3), инициализируются их веса с помощью nn.init.xavier\_uniform\_, применяемой к весам каждого слоя. После создания экземпляра нейронной сети (model) мы выводим инициализированные веса каждого слоя с помощью команды print. Обратите внимание, что веса будут отображаться в терминале.

1. **Гиперпараметры. Скорость обучения и размер батча.**

Гиперпараметры в машинном обучении — это параметры, которые определяют поведение алгоритма обучения, но не являются непосредственно оптимизируемыми в процессе обучения модели. Гиперпараметры задаются до начала обучения и влияют на процесс обучения и производительность модели.

Скорость обучения (learning rate) — это гиперпараметр, который определяет, насколько сильно обновляются веса модели в процессе обратного распространения ошибки. Он определяет величину шага, с которым обновляются веса. Высокая скорость обучения может привести к быстрой сходимости, но может также вызвать нестабильность и расходимость обучения. Низкая скорость обучения может привести к медленной сходимости или застреванию в локальном оптимуме. Выбор оптимальной скорости обучения зависит от конкретной задачи и архитектуры модели, и требуется некоторый экспериментирование и настройка для достижения хорошей производительности модели.

Размер батча (batch size) — это гиперпараметр, который определяет количество образцов данных, которые передаются модели за одну итерацию обучения. Больший размер батча может ускорить обучение, так как вычисления могут выполняться параллельно, и градиенты могут обновляться с меньшей частотой. Однако больший размер батча требует большего объема памяти, и обновление весов может быть менее частым, что может привести к менее стабильному обучению. Маленький размер батча может привести к более грубым обновлениям весов, но более стабильному обучению и лучшей обобщающей способности модели. Выбор размера батча зависит от доступных ресурсов, размера обучающего набора данных и конкретной задачи. Часто размер батча подбирают эмпирически или проводят эксперименты для определения оптимального значения.

Пример использования гиперпараметров скорости обучения и размера батча при обучении нейронной сети с использованием библиотеки PyTorch:

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

# Определение архитектуры нейронной сети

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.fc1 = nn.Linear(100, 50)

self.fc2 = nn.Linear(50, 10)

def forward(self, x):

x = torch.relu(self.fc1(x))

x = self.fc2(x)

return x

# Загрузка данных

train\_dataset = ...

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

# Создание модели

model = Net()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.001)

# Обучение модели

for epoch in range(10):

for inputs, labels in train\_loader:

# Обнуление градиентов

optimizer.zero\_grad()

# Прямой проход

outputs = model(inputs)

# Вычисление функции потерь

loss = criterion(outputs, labels)

# Обратное распространение ошибки и обновление весов

loss.backward()

optimizer.step()

В этом примере мы создаем нейронную сеть с двумя полносвязными слоями (nn.Linear) и активацией ReLU. Мы также определяем размер входных данных 100 и размер выходного слоя 10. Далее мы создаем загрузчик данных (DataLoader) с размером батча 32 для тренировочного набора данных. Затем мы создаем модель и определяем функцию потерь (nn.CrossEntropyLoss()) и оптимизатор (optim.SGD) с заданной скоростью обучения (lr=0.001). В цикле обучения мы обнуляем градиенты, выполняем прямой проход, вычисляем функцию потерь, выполняем обратное распространение ошибки и обновляем веса с помощью оптимизатора.

1. **Переобучение модели и регуляризация. Dropout.**

Переобучение модели — это явление, когда модель хорошо справляется с обучающими данными, но плохо обобщается на новые, ранее не виданные данные. Переобучение происходит, когда модель слишком подстраивается под особенности тренировочных данных и не обобщает их на более широкую популяцию данных.

Регуляризация — это методика, используемая для снижения переобучения модели. Она включает в модель дополнительные члены (регуляризаторы), которые контролируют сложность модели или ограничивают значения весов. Один из популярных методов регуляризации — это Dropout.

Dropout — это метод регуляризации, который случайным образом исключает («отключает») некоторые нейроны во время обучения модели. В процессе обучения, при каждом обновлении весов, каждый нейрон в определенном слое может быть исключен из сети с определенной вероятностью (обычно от 0,2 до 0,5). Таким образом, нейроны не могут полностью полагаться на другие нейроны и выучивают более устойчивые и обобщающие признаки. Применение Dropout помогает снизить переобучение, поскольку это заставляет модель обучаться на различных подмножествах нейронов, также помогает повысить обобщающую способность модели, особенно когда имеется ограниченное количество тренировочных данных, и предотвращает сильную зависимость между нейронами. Dropout также позволяет более эффективно использовать ресурсы обучения, так как он позволяет нейронам обучаться независимо. Вот пример использования Dropout в нейронной сети:

import torch

import torch.nn as nn

class MyModel(nn.Module):

def \_\_init\_\_(self):

super(MyModel, self).\_\_init\_\_()

self.fc1 = nn.Linear(64, 128)

self.dropout = nn.Dropout(0.5) # Применение Dropout с вероятностью 0.5

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

x = torch.relu(self.fc1(x))

x = self.dropout(x)

x = self.fc2(x)

return x

model = MyModel()

В этом примере мы создаем класс модели MyModel, который наследуется от nn.Module. Модель содержит два полносвязных (Linear) слоя и слой Dropout. Dropout применяется с вероятностью 0.5 между первым и вторым слоями. В методе forward определено прямое распространение данных через слои.

1. **Минбатчи – причина использования. Нормализация по мини-батчам.**

Мини-батчи (mini-batches) являются частями обучающего набора данных, на которых модель обновляет свои веса в процессе обучения. Вместо того, чтобы передавать все тренировочные данные в модель одновременно, данные разбиваются на небольшие части, называемые мини-батчами, и обновление весов происходит поэтапно на каждом мини-батче. Использование мини-батчей имеет несколько причин:

*1.* *Эффективность вычислений*: Работа с мини-батчами позволяет обрабатывать данные параллельно, что может значительно ускорить обучение модели. Вместо вычисления градиента на всем обучающем наборе данных одновременно, мы можем вычислить градиент только на текущем мини-батче, что является более эффективным с вычислительной точки зрения.

*2. Более стабильное обучение*: Работа с мини-батчами помогает сгладить шум в данных и делает обучение более стабильным. Вместо того, чтобы обновлять веса после каждого примера данных, мы обновляем их после каждого мини-батча. Это позволяет усреднить градиенты и уменьшить влияние отдельных шумных примеров.

Нормализация по мини-батчам (batch normalization) — это метод нормализации данных, который применяется к каждому мини-батчу данных перед их подачей в модель. Он позволяет центрировать и масштабировать данные внутри каждого мини-батча, чтобы обеспечить стабильность и улучшить процесс обучения модели. Процесс нормализации по мини-батчам включает в себя следующие шаги:

1. Вычисление среднего значения и дисперсии по каждому признаку внутри мини-батча.

2. Центрирование данных путем вычитания среднего значения.

3. Масштабирование данных путем деления на стандартное отклонение.

Это позволяет обеспечить нормализацию данных внутри каждого мини-батча и улучшить стабильность обучения модели. Нормализация по мини-батчам также имеет регуляризирующий эффект, который может помочь в борьбе с переобучением. Применение нормализации по мини-батчам в нейронных сетях помогает ускорить обучение, улучшить производительность модели и сделать процесс обучения более стабильным и предсказуемым.

Пример использования мини-батчей и нормализации по мини-батчам при обучении нейронной сети:

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader

# Определение нейронной сети

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.fc1 = nn.Linear(784, 64)

self.fc2 = nn.Linear(64, 10)

self.relu = nn.ReLU()

self.batchnorm = nn.BatchNorm1d(64) # Нормализация по мини-батчам

def forward(self, x):

x = self.fc1(x)

x = self.relu(x)

x = self.batchnorm(x) # Применение нормализации по мини-батчам

x = self.fc2(x)

return x

# Загрузка данных

train\_dataset = ... # Ваш тренировочный датасет

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

# Инициализация модели

model = Net()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(model.parameters(), lr=0.01)

# Обучение модели

for epoch in range(10):

for batch\_data, batch\_labels in train\_loader:

optimizer.zero\_grad()

# Преобразование данных в плоский вектор

batch\_data = batch\_data.view(-1, 784)

# Нормализация по мини-батчам

batch\_data\_normalized = (batch\_data - torch.mean(batch\_data, dim=0)) / torch.std(batch\_data, dim=0)

# Прямой проход

outputs = model(batch\_data\_normalized)

loss = criterion(outputs, batch\_labels)

# Обратное распространение

loss.backward()

optimizer.step()

В этом примере мы создаем нейронную сеть с использованием модулей PyTorch. Особое внимание уделяется использованию nn.BatchNorm1d для нормализации по мини-батчам. Во время обучения мы используем DataLoader для загрузки тренировочных данных в мини-батчах. Перед подачей мини-батча в модель мы выполняем нормализацию по мини-батчам, вычитая среднее значение и деля на стандартное отклонение каждого признака.

1. **Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.**

Задачи машинного обучения на изображениях представляют собой класс задач, связанных с анализом и обработкой изображений с использованием методов машинного обучения. Эти задачи включают распознавание объектов, классификацию изображений, сегментацию изображений, детекцию объектов, генерацию изображений и многое другое. Сверточные нейронные сети (Convolutional Neural Networks, CNN) являются одним из основных инструментов для работы с изображениями в машинном обучении.

Принцип работы сверточных сетей заключается в использовании операции свертки для выделения локальных особенностей и структур в изображениях. Сверточные слои в сверточной сети состоят из набора фильтров (ядер), которые скользят по входному изображению и применяются с использованием операции свертки. Каждый фильтр выделяет определенные признаки изображения, такие как границы, углы или текстуры, создавая карту активаций. Сверточные сети также используют операцию пулинга (pooling), которая уменьшает размерность карты активаций, сохраняя наиболее значимую информацию. Обычно используется операция субдискретизации (subsampling), такая как операция максимального пулинга (max pooling), которая выбирает максимальное значение из некоторой области.

Преимущества сверточных сетей при решении задач машинного обучения на изображениях включают:

1. *Иерархическое извлечение признаков:* Сверточные сети способны иерархически извлекать признаки из изображений на разных уровнях абстракции. Более низкие слои выделяют простые признаки, такие как границы и текстуры, а более высокие слои сочетают эти признаки для распознавания более сложных объектов и понятий.
2. *Разделение параметров:* Сверточные сети обладают свойством разделения параметров, что означает, что одни и те же параметры фильтров могут быть использованы для обработки разных областей изображения. Это позволяет сети эффективно работать с изображениями различного размера.
3. *Устойчивость к переносу:* Сверточные сети обладают свойством устойчивости к небольшим изменениям входных данных, таким как сдвиг, масштабирование или небольшие искажения. Это позволяет модели обобщать и лучше распознавать объекты, несмотря на некоторую вариабельность в данных.
4. *Параметры обучения:* Сверточные сети имеют много обучаемых параметров, которые позволяют модели адаптироваться к различным классам и объектам в изображениях. Параметры обучения сверточных сетей оптимизируются с использованием методов градиентного спуска, что позволяет модели автоматически настраивать свои параметры для достижения лучшей производительности.

Эти преимущества делают сверточные сети мощным инструментом для решения задач машинного обучения на изображениях, таких как классификация объектов, детектирование объектов, сегментация изображений и др.

Пример, который демонстрирует создание и обучение сверточной нейронной сети для классификации изображений:

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision.transforms as transforms

from torchvision.datasets import CIFAR10

from torch.utils.data import DataLoader

# Загрузка и предобработка данных

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

train\_dataset = CIFAR10(root='./data', train=True, download=True, transform=transform)

test\_dataset = CIFAR10(root='./data', train=False, download=True, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=64, shuffle=False)

# Определение модели сверточной нейронной сети

class CNN(nn.Module):

def \_\_init\_\_(self):

super(CNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, kernel\_size=3, stride=1, padding=1)

self.relu = nn.ReLU()

self.maxpool = nn.MaxPool2d(kernel\_size=2, stride=2)

self.conv2 = nn.Conv2d(16, 32, kernel\_size=3, stride=1, padding=1)

self.fc = nn.Linear(32 \* 8 \* 8, 10)

def forward(self, x):

x = self.conv1(x)

x = self.relu(x)

x = self.maxpool(x)

x = self.conv2(x)

x = self.relu(x)

x = self.maxpool(x)

x = x.view(x.size(0), -1)

x = self.fc(x)

return x

# Инициализация модели

model = CNN()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Обучение модели

num\_epochs = 10

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model.to(device)

for epoch in range(num\_epochs):

running\_loss = 0.0

for i, (inputs, labels) in enumerate(train\_loader):

inputs = inputs.to(device)

labels = labels.to(device)

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

if (i+1) % 100 == 0:

print(f'Epoch [{epoch+1}/{num\_epochs}], Step [{i+1}/{len(train\_loader)}], Loss: {running\_loss/100}')

running\_loss = 0.0

# Оценка модели на тестовых данных

model.eval()

correct = 0

total = 0

with torch.no\_grad():

for inputs, labels in test\_loader:

inputs = inputs.to(device)

labels = labels.to(device)

outputs = model(inputs)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

accuracy = 100 \* correct / total

print(f'Accuracy on the test set: {accuracy}%')

В этом примере мы используем библиотеку PyTorch для создания сверточной нейронной сети для классификации изображений из датасета CIFAR-10. Модель состоит из двух сверточных слоев с функцией активации ReLU, операции пулинга (max pooling) и полносвязного слоя для классификации. Мы также определяем функцию потерь CrossEntropyLoss и оптимизатор Adam для обучения модели. Модель обучается на тренировочном датасете в течение нескольких эпох с помощью градиентного спуска. Затем мы оцениваем точность модели на тестовом датасете.

1. **Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.**

Архитектура многослойной искусственной нейронной сети (ИНС) для распознавания изображений на основе сверточных сетей обычно состоит из нескольких типов слоев, которые позволяют модели извлекать иерархические признаки из изображений и классифицировать их. Вот общая структура такой архитектуры:

*1. Сверточные слои (Convolutional Layers):* Эти слои выполняют операцию свертки над входным изображением с использованием набора фильтров (ядер). Каждый фильтр выделяет определенные признаки изображения, такие как границы, углы или текстуры. Сверточные слои помогают модели выявлять локальные особенности изображения.

*2. Слой пулинга (Pooling Layer):* После каждого сверточного слоя обычно следует слой пулинга. Этот слой уменьшает пространственные размеры карты активаций, удаляя избыточные данные и сохраняя наиболее значимую информацию. Операция пулинга, такая как операция максимального пулинга (max pooling), выбирает максимальное значение из некоторой области.

*3. Полносвязные слои (Fully Connected Layers):* После нескольких сверточных слоев и слоев пулинга обычно добавляются полносвязные слои, которые объединяют выделенные признаки из предыдущих слоев для классификации изображений. Полносвязные слои имеют все пары нейронов соединены друг с другом.

*4. Слои активации (Activation Layers):* Между каждыми слоями в нейронной сети могут использоваться слои активации, которые вводят нелинейность в модель. Распространенные функции активации в сверточных сетях включают ReLU (Rectified Linear Unit), которая устанавливает все отрицательные значения на ноль, и softmax, которая преобразует выходные значения в вероятности для классификации.

*5. Выходной слой (Output Layer):* Выходной слой определяет формат выходных данных и обычно содержит количество нейронов, равное количеству классов в задаче классификации. Для задачи классификации изображений он часто использует функцию активации softmax для получения вероятностей принадлежности к каждому классу.

Примеры архитектур сверточных сетей для распознавания изображений включают LeNet-5, AlexNet, VGGNet, ResNet, Inception и другие. Эти архитектуры имеют различное количество слоев и параметров, и они были разработаны для улучшения производительности в задачах классификации изображений. Выбор конкретной архитектуры зависит от конкретной задачи и доступных ресурсов вычислительной мощности. Пример, который демонстрирует создание и обучение сверточной нейронной сети для классификации изображений:

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

# Загрузка и предобработка данных

transform = transforms.Compose([

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32, shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=32, shuffle=False, num\_workers=2)

# Определение архитектуры сверточной нейронной сети

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 64, 3)

self.conv2 = nn.Conv2d(64, 128, 3)

self.pool = nn.MaxPool2d(2, 2)

self.fc1 = nn.Linear(128 \* 6 \* 6, 256)

self.fc2 = nn.Linear(256, 10)

def forward(self, x):

x = self.pool(torch.relu(self.conv1(x)))

x = self.pool(torch.relu(self.conv2(x)))

x = x.view(-1, 128 \* 6 \* 6)

x = torch.relu(self.fc1(x))

x = self.fc2(x)

return x

net = Net()

# Определение функции потерь и оптимизатора

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# Обучение модели

for epoch in range(10):

running\_loss = 0.0

for i, data in enumerate(trainloader, 0):

inputs, labels = data

optimizer.zero\_grad()

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

if i % 2000 == 1999:

print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running\_loss / 2000))

running\_loss = 0.0

print('Finished training')

1. **Принципиальная логика обучения нейронной сети.**

Основная логика обучения нейронной сети заключается в итеративном процессе обновления весов сети на основе вычисленных градиентов. Процесс обучения включает следующие шаги:

*1. Инициализация модели:* Создание архитектуры нейронной сети, определение числа слоев, типов слоев и их размерности. Параметры модели, включая веса и смещения, инициализируются случайными значениями.

*2. Прямой проход:* Входные данные передаются через слои нейронной сети от входного слоя к выходному слою. Каждый слой выполняет операции, например, умножение на веса и применение функции активации, чтобы получить прогнозы модели.

*3. Вычисление функции потерь:* Сравнение прогнозов модели с правильными метками для оценки ошибки. Функция потерь измеряет, насколько модель ошибается в предсказании.

*4. Обратное распространение ошибки:* Распространение ошибки назад через сеть для вычисления градиентов функции потерь по весам сети. Градиенты показывают направление и величину изменения весов, необходимое для улучшения прогнозов модели.

*5. Обновление весов:* Используя градиенты, оптимизационный алгоритм (например, стохастический градиентный спуск) обновляет веса сети, смещения и другие параметры, чтобы минимизировать функцию потерь. Обновление происходит путем вычитания градиента из текущих значений весов.

*6. Повторение шагов:* Процесс прямого и обратного проходов повторяется на нескольких эпохах (полных проходах через все обучающие данные) для того, чтобы модель могла лучше настроить свои параметры и улучшить предсказания.

*7. Оценка модели:* После завершения обучения модель оценивается на отложенной выборке или тестовых данных для оценки ее производительности. Это позволяет оценить точность, полноту, F-меру или другие метрики, в зависимости от задачи.

*8. Настройка гиперпараметров:* Гиперпараметры, такие как скорость обучения, количество слоев, количество нейронов и другие, могут быть настроены для достижения лучших результатов.

*9. Инференс:* Обученная модель может быть использована для предсказания значений на новых данных, которые не были использованы во время обучения.

1. **Приемы для глубокого обучения на небольших наборах изображений.**

Глубокое обучение на небольших наборах изображений может быть сложной задачей, поскольку глубокие модели обычно требуют большого количества размеченных данных для эффективного обучения. Однако существуют несколько приемов, которые могут помочь улучшить обучение модели на ограниченном количестве данных. Это некоторые из них:

1. *Аугментация данных (Data Augmentation):* Это техника, которая позволяет генерировать новые тренировочные примеры на основе существующих данных путем применения различных преобразований, таких как повороты, масштабирование, отражение и сдвиги. Это помогает увеличить разнообразие данных и предотвратить переобучение. Пример:

import torchvision.transforms as transforms

# Определение преобразований аугментации данных

transform = transforms.Compose([

transforms.RandomHorizontalFlip(),

transforms.RandomRotation(10),

transforms.RandomResizedCrop(224),

transforms.ToTensor(),

])

# Применение аугментации к изображениям

augmented\_images = []

for image in images:

augmented\_image = transform(image)

augmented\_images.append(augmented\_image)

1. *Перенос обучения (Transfer Learning):* Вместо обучения модели с нуля на небольшом наборе данных, можно использовать предварительно обученные модели, которые были обучены на больших наборах данных, и настроить их на своем небольшом наборе данных. Предварительно обученные модели содержат в себе общую способность извлекать признаки из изображений, которая может быть полезной для задачи классификации на вашем небольшом наборе данных. Пример:

import torchvision.models as models

# Загрузка предварительно обученной модели

model = models.resnet50(pretrained=True)

# Замораживание параметров предварительно обученной модели

for param in model.parameters():

param.requires\_grad = False

# Замена последнего полносвязного слоя для новой задачи

num\_classes = 10

model.fc = nn.Linear(model.fc.in\_features, num\_classes)

# Обучение модели на небольшом наборе данных

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()

for epoch in range(num\_epochs):

for images, labels in dataloader:

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

1. *Регуляризация (Regularization):* Регуляризация помогает предотвратить переобучение модели на небольшом наборе данных. Вы можете использовать различные методы регуляризации, такие как Dropout, L1 и L2 регуляризация, чтобы ограничить сложность модели и улучшить ее обобщающую способность. Пример:

import torch.nn as nn

# Определение модели с Dropout слоями

class MyModel(nn.Module):

def \_\_init\_\_(self):

super(MyModel, self).\_\_init\_\_()

self.fc1 = nn.Linear(784, 256)

self.dropout1 = nn.Dropout(0.5)

self.fc2 = nn.Linear(256, 128)

self.dropout2 = nn.Dropout(0.5)

self.fc3 = nn.Linear(128, 10)

def forward(self, x):

x = x.view(x.size(0), -1)

x = F.relu(self.fc1(x))

x = self.dropout1(x)

x = F.relu(self.fc2(x))

x = self.dropout2(x)

x = self.fc3(x)

return x

# Инициализация модели

model = MyModel()

# Обучение модели с регуляризацией

optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight\_decay=0.001)

criterion = nn.CrossEntropyLoss()

for epoch in range(num\_epochs):

for images, labels in dataloader:

optimizer.zero\_grad()

outputs = model(images)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

1. *Использование предобработанных признаков (Pretrained Features):* Вместо использования пикселей изображений как входных данных для модели, вы можете использовать предварительно извлеченные признаки, полученные с помощью предварительно обученных сверточных сетей. Это позволяет снизить количество параметров модели и требуемую размеченную информацию. Пример:

import torchvision.models as models

# Загрузка предварительно обученной модели

model = models.resnet50(pretrained=True)

# Получение признаков изображений

features = []

for image in images:

feature = model(image.unsqueeze(0))

features.append(feature)

# Использование признаков для классификации

# (добавить собственную логику для обучения модели на полученных признаках)

1. *Генерация синтетических данных (Synthetic Data Generation):* Если у вас есть ограниченное количество данных, вы можете использовать методы генерации синтетических данных, таких как генеративные модели (например, GAN) или алгоритмы увеличения разрешения (например, SRGAN), чтобы создать дополнительные тренировочные примеры. Пример:

# Генерация синтетических изображений с помощью GAN

for i in range(num\_samples):

z = torch.randn(batch\_size, latent\_dim)

fake\_images = generator(z)

# Использование сгенерированных изображений для обучения модели

1. *Однократное обучение (One-Shot Learning):* Для задач с очень небольшим количеством обучающих примеров можно использовать методы однократного обучения, которые позволяют модели обучаться на основе одного или нескольких примеров из каждого класса.
2. **Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.**

Сверточные сети (Convolutional Neural Networks, CNN) являются эффективным инструментом для обработки изображений. Они состоят из различных слоев, включая сверточные слои и слои пулинга, которые выполняют ключевые операции при анализе изображений.

Схема работы сверточного слоя:

*1. Входные данные:* Сверточный слой принимает входные данные в виде трехмерного тензора (высота x ширина x количество каналов). Например, для цветных изображений с тремя каналами RGB, входной тензор будет иметь размерность (высота x ширина x 3).

*2. Фильтры:* Сверточный слой содержит набор фильтров (ядер), каждый из которых представляет собой небольшую матрицу весов. Фильтры применяются к входным данным, перемещаясь по изображению и вычисляя свертку.

*3. Свертка:* Для каждого фильтра выполняется свертка, которая представляет собой перемещение фильтра по изображению и поэлементное умножение весов фильтра на соответствующие пиксели входных данных. Затем результаты умножений суммируются, образуя одно число (пиксель) в выходном тензоре.

*4. Применение функции активации:* На выходе из сверточного слоя применяется нелинейная функция активации, такая как ReLU (Rectified Linear Unit), которая добавляет нелинейность к модели.

*5. Выходные данные:* Результатом работы сверточного слоя является трехмерный тензор, называемый картой признаков. Его размерность зависит от количества фильтров и размеров входного изображения.

Слой пулинга (Pooling):

После сверточного слоя обычно следует слой пулинга, который выполняет субдискретизацию карты признаков путем уменьшения ее размерности. Основная цель пулинга состоит в сокращении пространственных размеров карты признаков, сохраняя при этом важные особенности. Наиболее распространенным типом пулинга является Max Pooling, который выбирает максимальное значение внутри каждого окна пулинга. Другие типы пулинга включают среднее значение (Average Pooling) и L2-норму (L2 Pooling).

Гиперпараметры сверточных слоев:

*1. Padding:* Параметр, определяющий заполнение (padding) входных данных вокруг краев перед выполнением свертки. Он может принимать значения "valid" (без заполнения) или "same" (заполнение, чтобы сохранить размеры входных данных).

*2. Kernel Size:* Размер фильтра (ядра) свертки. Он определяет размер окна, перемещающегося по изображению, и может быть задан в виде одного числа (квадратное ядро) или в виде кортежа (прямоугольное ядро).

*3. Stride:* Шаг перемещения окна свертки по изображению. Определяет, насколько пикселей перемещается окно при выполнении свертки.

*4. Dilation:* Параметр, определяющий расширение (dilation) фильтра. Он контролирует интервал между значениями фильтра и влияет на размер области, которую фильтр анализирует на каждом шаге.

Пример использования сверточных слоев и слоя пулинга в модели на основе фреймворка PyTorch:

import torch

import torch.nn as nn

# Определение модели

class CNNModel(nn.Module):

def \_\_init\_\_(self):

super(CNNModel, self).\_\_init\_\_()

# Сверточный слой

self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=16, kernel\_size=3, stride=1, padding=1)

# Слой пулинга

self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)

# Полносвязные слои

self.fc1 = nn.Linear(16 \* 16 \* 16, 128)

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

# Проход через сверточный слой

x = self.conv1(x)

x = nn.ReLU()(x)

# Проход через слой пулинга

x = self.pool(x)

# Выравнивание для подачи в полносвязные слои

x = x.view(-1, 16 \* 16 \* 16)

# Проход через полносвязные слои

x = self.fc1(x)

x = nn.ReLU()(x)

x = self.fc2(x)

return x

# Создание экземпляра модели

model = CNNModel()

# Входные данные размером (batch\_size x channels x height x width)

inputs = torch.randn(32, 3, 32, 32)

# Проход через модель

outputs = model(inputs)

# Вывод размерности выходных данных

print(outputs.size())

В этом примере мы определяем класс CNNModel, который наследуется от nn.Module в PyTorch. Модель содержит сверточный слой (conv1), слой пулинга (pool) и полносвязные слои (fc1, fc2). В методе forward мы указываем последовательность операций, включая свертку, активацию, пулинг, выравнивание и полносвязные слои.Затем мы создаем экземпляр модели и передаем входные данные размером (32, 3, 32, 32) через модель. Размерность выходных данных выводится с помощью print(outputs.size()).

1. **Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.**

Задачи обработки текста включают в себя ряд задач, связанных с анализом и пониманием текстовых данных. Некоторые из таких задач включают дистрибутивную семантику, матрицу совместной встречаемости и представление слов в виде векторов малой размерности:

*1. Дистрибутивная семантика:* Дистрибутивная семантика изучает связь между словами на основе их контекста в больших текстовых корпусах. Она основывается на предположении, что слова, которые появляются в похожих контекстах, имеют схожую семантику. На основе дистрибутивной семантики можно строить модели, которые представляют слова в виде векторов, называемых эмбеддингами, где семантически близкие слова имеют близкие векторные представления.

*2. Матрица совместной встречаемости:* Матрица совместной встречаемости (Co-occurrence Matrix) представляет собой матрицу, в которой каждый элемент отражает частоту совместного появления двух слов в контексте. Каждое слово в текстовом корпусе представлено в виде уникального индекса, и значения в матрице совместной встречаемости отражают частоты совместного появления пар слов.

*3. Представление слов в виде векторов малой размерности:* Для эффективной обработки текстовых данных в машинном обучении часто используется представление слов в виде векторов малой размерности. Такие векторы называются словными эмбеддингами и позволяют представить семантическое значение слова в числовом виде. Одним из популярных методов получения словных эмбеддингов является Word2Vec, который использует нейронные сети для обучения векторных представлений на основе контекста слов.

Сочетание этих подходов позволяет моделям обрабатывать и понимать текстовые данные. Например, представление слов в виде векторов малой размерности позволяет измерять семантическую близость между словами или использовать эти векторы как признаки для классификации или кластеризации текстовых данных. Матрица совместной встречаемости может быть использована для анализа частоты и семантических связей между словами в текстовом корпусе.

1. **Word2vec: модель CBOW.**

**Word2Vec** — это алгоритм машинного обучения, разработанный для создания векторных представлений слов на основе их семантического контекста в больших текстовых корпусах. Векторные представления, полученные с помощью Word2Vec, позволяют компьютеру понимать смысл слов и выявлять семантические отношения между ними.

Модель CBOW (Continuous Bag-of-Words) в Word2Vec является одной из двух основных моделей для создания векторных представлений слов и основывается на принципе предсказания текущего слова по контексту окружающих его слов. Она стремится минимизировать разницу между предсказанным словом и фактическим целевым словом. В обучающей выборке входными данными являются контекстные слова, а выходными данными - текущее слово. Каждое слово представляется в виде one-hot вектора, где все элементы равны нулю, кроме одного элемента, соответствующего индексу слова в словаре, который равен единице.

CBOW использует простую нейронную сеть с одним скрытым слоем. При обучении CBOW модели используется окно фиксированного размера вокруг каждого слова, чтобы определить контекст. Входной слой имеет размерность, равную размеру окна контекста, а выходной слой имеет размерность, равную размеру словаря. Скрытый слой содержит небольшое количество нейронов и служит для сжатия информации из входного слоя. Входной слой принимает векторные представления контекстных слов и передает их в скрытый слой. Затем скрытый слой объединяет эти векторы и передает информацию в выходной слой, который предсказывает целевое слово и содержит его векторное представление.

Модель CBOW в Word2Vec обладает простотой и эффективностью. Она особенно хорошо работает для частотных слов и проявляет себя лучше в задачах, где контекст более важен, чем точное положение слова. Преимущество модели CBOW заключается в ее скорости обучения и способности обрабатывать большие текстовые корпуса, поскольку она обрабатывает контекстные слова параллельно. Кроме того, модель CBOW часто демонстрирует хорошие результаты на задачах семантического анализа и нахождения аналогий между словами. После обучения модели CBOW можно использовать полученные векторные представления слов для различных задач обработки естественного языка, таких как кластеризация слов, классификация текстов, извлечение информации и т. д.

Пример использования модели CBOW на основе библиотеки PyTorch:

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset

# Определение модели CBOW

class CBOW(nn.Module):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, context\_size):

super(CBOW, self).\_\_init\_\_()

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.linear = nn.Linear(context\_size \* embedding\_dim, vocab\_size)

def forward(self, inputs):

embedded = self.embedding(inputs).view(1, -1)

out = self.linear(embedded)

log\_probs = nn.functional.log\_softmax(out, dim=1)

return log\_probs

# Определение набора данных для обучения модели

class CBOWDataset(Dataset):

def \_\_init\_\_(self, data, context\_size):

self.data = data

self.context\_size = context\_size

def \_\_getitem\_\_(self, index):

start = index - self.context\_size

end = index + self.context\_size + 1

context = self.data[start:end]

target = self.data[index]

return torch.tensor(context), torch.tensor(target)

def \_\_len\_\_(self):

return len(self.data)

# Обучение модели CBOW

def train\_cbow(data, vocab\_size, embedding\_dim, context\_size, epochs, lr):

model = CBOW(vocab\_size, embedding\_dim, context\_size)

criterion = nn.NLLLoss()

optimizer = optim.SGD(model.parameters(), lr=lr)

dataset = CBOWDataset(data, context\_size)

dataloader = DataLoader(dataset, batch\_size=1, shuffle=True)

for epoch in range(epochs):

total\_loss = 0

for context, target in dataloader:

optimizer.zero\_grad()

log\_probs = model(context)

loss = criterion(log\_probs, target)

loss.backward()

optimizer.step()

total\_loss += loss.item()

print(f"Epoch {epoch+1}/{epochs}, Loss: {total\_loss}")

return model

1. **Многослойные сети. Граф потока вычислений.**

Многослойные сети (multilayer networks) представляют собой нейронные сети, состоящие из нескольких слоев нейронов, которые обрабатывают входные данные последовательно, в отличие от однослойных сетей. Каждый слой в многослойной сети обычно состоит из набора нейронов, которые связаны с нейронами в предыдущем и следующем слоях. Нейроны в одном слое получают входные данные от предыдущего слоя, вычисляют взвешенную сумму входов и применяют функцию активации для генерации выхода. Выходы слоя передаются в следующий слой в качестве входных данных.

Граф потока вычислений (computational graph) представляет собой графическое представление вычислительной структуры многослойной сети. Он показывает поток данных от входных данных к выходным данным через слои нейронной сети. Каждый слой представлен узлом в графе, а связи между слоями показывают направление потока данных. Граф потока вычислений удобен для визуализации и понимания структуры нейронной сети. Он также является основой для вычислений и оптимизации во время обучения модели. В графе потока вычислений определяются входные данные, веса и смещения слоев, а также операции для преобразования данных в каждом слое.

Преимущество многослойных сетей и графов потока вычислений заключается в их способности моделировать сложные функции и обрабатывать сложные данные, такие как изображения, звук, текст и другие. Благодаря возможности последовательного применения преобразований на разных уровнях абстракции многослойные сети обладают способностью извлекать иерархические признаки из данных и делать более сложные выводы и предсказания. Пример создания многослойной нейронной сети с использованием библиотеки PyTorch и построения графа потока вычислений:

import torch

import torch.nn as nn

# Определение класса модели

class MultiLayerNet(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

super(MultiLayerNet, self).\_\_init\_\_()

# Определение слоев модели

self.fc1 = nn.Linear(input\_size, hidden\_size)

self.fc2 = nn.Linear(hidden\_size, hidden\_size)

self.fc3 = nn.Linear(hidden\_size, output\_size)

self.relu = nn.ReLU()

def forward(self, x): # Построение графа потока вычислений

x = self.relu(self.fc1(x))

x = self.relu(self.fc2(x))

x = self.fc3(x)

return x

# Создание экземпляра модели

input\_size = 784

hidden\_size = 64

output\_size = 10

model = MultiLayerNet(input\_size, hidden\_size, output\_size)

# Вывод структуры модели

print(model)

1. **Word2vec: модель Skip-Gram.**

Word2Vec — это алгоритм машинного обучения, разработанный для создания векторных представлений слов на основе их семантического контекста в больших текстовых корпусах. Векторные представления, полученные с помощью Word2Vec, позволяют компьютеру понимать смысл слов и выявлять семантические отношения между ними.

Модель Skip-Gram в Word2Vec является одной из двух основных моделей для создания векторных представлений слов и основана на принципе предсказания контекста слова. Она предсказывает контекстные слова на основе заданного целевого слова. То есть, задачей Skip-Gram является обучение модели на основе обучающего примера, который состоит из текущего слова и одного или нескольких окружающих слов, которые служат контекстом. Каждое слово представляется в виде one-hot вектора, где все элементы равны нулю, кроме одного элемента, соответствующего индексу слова в словаре, который равен единице.

Skip-Gram использует простую нейронную сеть с одним скрытым слоем. Входной слой имеет размерность, равную размеру словаря, а выходной слой имеет размерность, равную размеру окна контекста. Скрытый слой содержит небольшое количество нейронов (например, 100–300) и служит для сжатия информации из входного слоя. Входной слой принимает векторное представление целевого слова и передает его в скрытый слой. Затем скрытый слой преобразует их в скрытое представление и передает в выходной слой. Выходной слой предсказывает контекстные слова и содержит их векторные представления. После этого происходит обновление весов модели на основе разницы между предсказанными и фактическими контекстными словами.

Векторные представления, полученные с помощью модели Skip-Gram, имеют интересное свойство. Семантически близкие слова имеют близкие векторы, что позволяет выполнять алгебраические операции с векторами слов. Например, можно выполнить операцию "король" - "мужчина" + "женщина" и получить вектор, который близок к вектору слова "королева". Это свойство векторных представлений позволяет выполнять аналогии и поиск близких слов.

Skip-Gram модель в Word2Vec имеет преимущество в работе с редкими словами и в задачах, где точное положение слова имеет большое значение. Она позволяет получить более точные векторные представления слов, но может потребовать больше вычислительных ресурсов и времени для обучения по сравнению со второй моделью в Word2Vec – CBOW, так как требует предсказания нескольких контекстных слов для каждого целевого слова. Еще одно преимущество модели Skip-Gram заключается в ее способности работать с богатыми семантически контекстами. Она может лучше представлять семантически близкие слова и отношения между ними. Векторы, полученные с помощью Skip-Gram, часто используются для решения задач, таких как поиск похожих слов, классификация слов, семантическая аналогия и т. д.

Пример использования модели Skip-Gram в библиотеке PyTorch:

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset

# Определение модели Skip-Gram

class SkipGram(nn.Module):

def \_\_init\_\_(self, vocab\_size, embedding\_dim):

super(SkipGram, self).\_\_init\_\_()

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.linear = nn.Linear(embedding\_dim, vocab\_size)

def forward(self, target):

embedded = self.embedding(target)

out = self.linear(embedded)

log\_probs = nn.functional.log\_softmax(out, dim=1)

return log\_probs

# Определение набора данных для обучения модели

class SkipGramDataset(Dataset):

def \_\_init\_\_(self, data, window\_size):

self.data = data

self.window\_size = window\_size

def \_\_getitem\_\_(self, index):

target = self.data[index]

start = index - self.window\_size

end = index + self.window\_size + 1

context = self.data[start:index] + self.data[index+1:end]

return torch.tensor(target), torch.tensor(context)

def \_\_len\_\_(self):

return len(self.data)

# Обучение модели Skip-Gram

def train\_skipgram(data, vocab\_size, embedding\_dim, window\_size, epochs, lr):

model = SkipGram(vocab\_size, embedding\_dim)

criterion = nn.NLLLoss()

optimizer = optim.SGD(model.parameters(), lr=lr)

dataset = SkipGramDataset(data, window\_size)

dataloader = DataLoader(dataset, batch\_size=1, shuffle=True)

for epoch in range(epochs):

total\_loss = 0

for target, context in dataloader:

optimizer.zero\_grad()

log\_probs = model(target)

loss = criterion(log\_probs, context)

loss.backward()

optimizer.step()

total\_loss += loss.item()

print(f"Epoch {epoch+1}/{epochs}, Loss: {total\_loss}")

return model

1. **Механизм Attention. Пример использования Attentinon.**

Механизм Attention (внимание) — это механизм, который позволяет нейронным сетям фокусироваться на определенных частях входных данных или контекста, присваивая им веса или значимость в процессе обработки. Он позволяет модели сосредоточиться на наиболее важных частях входных данных при выполнении задачи. Основная идея механизма Attention заключается в том, чтобы вместо того, чтобы кодировать всю информацию о последовательности в фиксированный вектор (как в случае с рекуррентными нейронными сетями), модель могла динамически обращаться к разным частям входных данных с разной степенью важности. Принцип работы механизма Attention следующий:

Для каждого элемента входной последовательности вычисляется вес, отражающий его важность или релевантность для текущего контекста. Это делается путем применения функции внимания к текущему состоянию модели и элементу последовательности. Веса могут быть вычислены различными способами, например, с использованием многослойного персептрона или скалярного произведения. Вычисленные веса нормализуются, обычно с помощью функции Softmax, чтобы сумма всех весов была равна 1. Каждый элемент входной последовательности умножается на его соответствующий вес, чтобы выделить его вклад в общий контекст. Выходные значения складываются или конкатенируются, чтобы получить итоговое представление или контекст.

Пример использования механизма Attention в задачах обработки текста:

import torch

import torch.nn as nn

class Attention(nn.Module):

def \_\_init\_\_(self, hidden\_size):

super(Attention, self).\_\_init\_\_()

self.hidden\_size = hidden\_size

self.attention\_weights = nn.Linear(hidden\_size \* 2, 1)

def forward(self, decoder\_hidden, encoder\_outputs):

seq\_len = encoder\_outputs.size(0)

# Повторяем скрытое состояние декодера для каждого временного шага кодировщика

decoder\_hidden = decoder\_hidden.expand(seq\_len, -1, -1)

# Конкатенируем скрытое состояние декодера и выходы кодировщика

concat = torch.cat((decoder\_hidden, encoder\_outputs), dim=2)

# Вычисляем веса внимания

attention\_scores = self.attention\_weights(concat)

attention\_weights = torch.softmax(attention\_scores, dim=0)

# Вычисляем взвешенную сумму выходов кодировщика

context\_vector = torch.sum(attention\_weights \* encoder\_outputs, dim=0)

return context\_vector, attention\_weights

1. **Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.**

**Рекуррентная нейронная сеть (RNN)** — это тип нейронных сетей, особенностью которого является наличие обратных связей, позволяющих нейросети использовать информацию из предыдущих шагов для обработки текущего входа. Каждый рекуррентный слой в RNN имеет скрытое состояние, которое передается от одного временного шага к другому. Это делает RNN особенно полезными для работы с последовательными или временными данными, где контекст и зависимости между элементами последовательности играют важную роль. Рекуррентные слои могут использовать различные типы ячеек, такие как простая рекуррентная ячейка (Simple RNN), LSTM (Long Short-Term Memory) или GRU (Gated Recurrent Unit), для моделирования зависимостей разной сложности.

Принципы обучения рекуррентных нейронных сетей:

*• Предобработка данных:* важным аспектом обучения RNN является предобработка данных. Например, текстовые данные могут быть представлены в виде последовательности слов или символов, которые должны быть закодированы числами. Также может потребоваться нормализация и масштабирование данных для улучшения процесса обучения.

*• Прямое распространение (проход):* на каждом шаге времени RNN принимает входные данные и передает их через слои с рекуррентными связями. Каждый шаг времени имеет свое скрытое состояние, которое является результатом обработки текущего входа и предыдущего скрытого состояния. Выходные значения могут быть получены на каждом шаге времени или только на последнем шаге.

*• Обратное распространение ошибки (Backpropagation):* после прямого распространения происходит вычисление функции потерь между предсказанным результатом и фактическим значением. Затем, во время обратного распространения ошибки, градиенты вычисляются в обратном порядке по времени, что позволяет обновить веса RNN и улучшить качество предсказания на последующих шагах времени.

Сложности применения рекуррентных нейронных сетей:

*• Обработка последовательностей переменной длины:* если данные имеют переменную длину, например текст различной длины, RNN способна обрабатывать последовательности переменной длины, но требуется предварительная обработка данных. В некоторых случаях используется заполнение (padding), чтобы все последовательности имели одинаковую длину. В других случаях используется механизм маскировки (masking), чтобы учитывать только активные элементы последовательности.

*• Ограниченная память:* Рекуррентные нейронные сети имеют ограниченную память и могут иметь проблемы с моделированием долгосрочных зависимостей, так как скрытое состояние обычно имеет фиксированный размер. Это особенно важно при работе с длинными последовательностями, где информация из начала последовательности может быть затерта.

*• Проблема затухания/взрыва градиента:* во время обратного распространения градиенты могут снижаться или увеличиваться экспоненциально, особенно при работе с глубокими рекуррентными сетями или длинными последовательностями. Это происходит из-за повторного умножения градиента на веса RNN во время обратного распространения. Это может затруднить обучение и привести к проблемам с сохранением долгосрочных зависимостей. Для решения этой проблемы были предложены различные методы, такие как обрезка градиента (gradient clipping) и рекуррентные ячейки, такие как LSTM (Long Short-Term Memory) и GRU (Gated Recurrent Unit), которые имеют специальную структуру для управления потоком градиента.

*• Вычислительная сложность:* RNN требуют больших вычислительных ресурсов из-за последовательной обработки данных и вычислений на каждом временном шаге (операции над матрицами и вычисления градиентов). Обучение RNN на больших наборах данных или с длинными последовательностями может быть вычислительно затратным и требовать мощных аппаратных ресурсов.

*• Зависимость от контекста:* RNN могут быть чувствительны к длине контекста и не всегда хорошо работать с долгосрочными зависимостями. Для решения этой проблемы могут потребоваться более сложные архитектуры RNN, такие как LSTM и GRU, или комбинации RNN с другими типами нейронных сетей, такими как сверточные нейронные сети.

Пример использования рекуррентной нейронной сети (LSTM) для задачи предсказания текста на основе последовательности слов:

import torch

import torch.nn as nn

import torch.optim as optim

from torch.utils.data import DataLoader, Dataset

# Определение модели LSTM

class LSTMModel(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, num\_layers, output\_size):

super(LSTMModel, self).\_\_init\_\_()

self.hidden\_size = hidden\_size

self.num\_layers = num\_layers

self.lstm = nn.LSTM(input\_size, hidden\_size, num\_layers, batch\_first=True)

self.fc = nn.Linear(hidden\_size, output\_size)

def forward(self, x):

h0 = torch.zeros(self.num\_layers, x.size(0), self.hidden\_size).to(x.device)

c0 = torch.zeros(self.num\_layers, x.size(0), self.hidden\_size).to(x.device)

out, \_ = self.lstm(x, (h0, c0))

out = self.fc(out[:, -1, :])

return out

# Определение набора данных для обучения модели

class TextDataset(Dataset):

def \_\_init\_\_(self, data, sequence\_length):

self.data = data

self.sequence\_length = sequence\_length

def \_\_getitem\_\_(self, index):

x = self.data[index:index+self.sequence\_length]

y = self.data[index+self.sequence\_length]

return torch.tensor(x), torch.tensor(y)

def \_\_len\_\_(self):

return len(self.data) - self.sequence\_length

# Обучение модели LSTM

def train\_lstm(data, input\_size, hidden\_size, num\_layers, output\_size, sequence\_length, epochs, lr):

model = LSTMModel(input\_size, hidden\_size, num\_layers, output\_size)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=lr)

dataset = TextDataset(data, sequence\_length)

dataloader = DataLoader(dataset, batch\_size=1, shuffle=True)

for epoch in range(epochs):

total\_loss = 0

for x, y in dataloader:

optimizer.zero\_grad()

out = model(x)

loss = criterion(out, y)

loss.backward()

optimizer.step()

total\_loss += loss.item()

print(f"Epoch {epoch+1}/{epochs}, Loss: {total\_loss}")

return model

1. **Модуль LSTM.**

Модуль LSTM (Long Short-Term Memory) является основным компонентом рекуррентных нейронных сетей (RNN), предназначенным для решения проблемы затухающего градиента и обработки долгосрочных зависимостей в последовательных данных. LSTM-модуль вводит специальный механизм управления памятью, который помогает сети запоминать и забывать информацию на основе контекста и заданной цели. Основная идея LSTM заключается в том, что он использует специальные блоки, называемые ячейками памяти (Memory cell). Ячейка памяти хранит и обновляет информацию с течением времени. Она может сохранять важные долгосрочные зависимости, что позволяет сети лучше работать с длинными последовательностями.

LSTM имеет скрытое (клеточное) состояние (Cell state), которое хранит и передает информацию на протяжении всей последовательности данных, позволяя моделировать долгосрочные зависимости. Скрытое (клеточное) состояние может быть представлено как вертикальная линия, которая протягивается через все временные шаги LSTM.

Узлы (Gates) в LSTM используются для контроля потока информации внутри ячейки. Они являются механизмом фильтрации и регулирования потока данных, который проходит через LSTM. Они позволяют LSTM определять, какую информацию следует сохранить, забыть или передать дальше. Есть три основных типа узлов:

*• Входной узел (Input gate)* принимает входные данные и предыдущее скрытое состояние, которые проходят через сигмоидную функцию активации, и определяет, какая информация будет добавлена в скрытое состояние ячейки.

*• Забывающий узел (Forget gate)* определяет, какая информация в скрытом состоянии ячейки должна быть забыта. Он принимает на вход текущий вход и предыдущее скрытое состояние, прогоняет их через сигмоидную функцию активации и генерирует значения от 0 до 1 для каждой информации в клеточном состоянии. Значение 0 означает полное забывание, а значение 1 - полное сохранение информации.

*• Выходной узел (Output gate)* принимает входные данные и предыдущее скрытое состояние, которые проходят через сигмоидную функцию, и определяет, какая информация будет передана на выход.

Так же в ячейке памяти есть узел обновления состояния (candidate cell state), который определяет новое скрытое состояние ячейки памяти на основе входных данных, предыдущего скрытого состояния и входного узла. В качестве функции активации он использует гиперболический тангенс при создании нового состояния.

Преимущества использования LSTM:

*• Предотвращение градиентного затухания и взрыва:* LSTM решает проблему градиентного затухания и взрыва, которая возникает при обучении RNN на длинных последовательностях данных. Благодаря узлам (Gates) LSTM может контролировать поток градиента и предотвращать его нестабильность.

*• Способность обрабатывать долгосрочные зависимости:* благодаря своей архитектуре, LSTM может запоминать информацию на протяжении долгого времени, что делает его особенно полезным для задач, требующих моделирования долгосрочных зависимостей в данных.

LSTM является мощным инструментом для обработки последовательных данных, позволяет преодолеть некоторые сложности, связанные с обработкой долгосрочных зависимостей, и широко используется в различных областях, таких как обработка естественного языка, машинный перевод, распознавание речи, генерация текста, анализ и прогнозирование временных рядов и других задачах, где важны долгосрочные зависимости в данных.

Пример использования модуля LSTM в PyTorch:

import torch

import torch.nn as nn

input\_size = 10

hidden\_size = 20

sequence\_length = 5

batch\_size = 3

# Создание модуля LSTM

lstm = nn.LSTM(input\_size, hidden\_size)

# Входные данные размерности (sequence\_length, batch\_size, input\_size)

inputs = torch.randn(sequence\_length, batch\_size, input\_size)

# Инициализация скрытого состояния и ячеек памяти LSTM

h0 = torch.zeros(1, batch\_size, hidden\_size)

c0 = torch.zeros(1, batch\_size, hidden\_size)

# Применение LSTM к входным данным

outputs, (hn, cn) = lstm(inputs, (h0, c0))

print(outputs.shape) # Размерность выхода: (sequence\_length, batch\_size, hidden\_size)

print(hn.shape) # Размерность последнего скрытого состояния: (1, batch\_size, hidden\_size)

print(cn.shape) # Размерность последней ячейки памяти: (1, batch\_size, hidden\_size)

1. **Модель BERT.**

BERT (Bidirectional Encoder Representations from Transformers) - это модель глубокого обучения, представленная в 2018 году в статье "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". BERT стал одной из наиболее влиятельных моделей в области обработки естественного языка (Natural Language Processing, NLP) и достиг значительных результатов в различных задачах, таких как классификация текста, анализ тональности, извлечение информации, вопросно-ответные системы и машинный перевод.

BERT – двунаправленная языковая модель, основанная на архитектуре трансформер, предназначенная для предобучения языковых представлений с целью их последующего применения в широком спектре задач обработки естественного языка. BERT основан на архитектуре Transformer, которая позволяет модели обрабатывать контексты длинных последовательностей с помощью механизма внимания (Attention). Архитектура Transformer позволяет модели BERT эффективно улавливать зависимости между словами в предложении. BERT обрабатывает тексты в обоих направлениях, что позволяет модели получать контекстуальные представления слов, учитывая как предшествующий, так и последующий контекст. Это помогает модели лучше понимать зависимости между словами в предложении.

BERT-модель предварительно обучена без учителя на на двух NLP-задачах:

• BERT использует задачу генерации пропущенного токена (masked language modeling) во время предварительного обучения. На вход BERT подаются токенизированные пары предложений, в которых некоторые токены скрыты. Таким образом, модель обучается генерировать пропущенные токены (слова или отдельные токены). Это позволяет модели учиться понимать контекст и смысл слов в предложении.

• BERT также использует задачу предсказания следующего предложения (next sentence prediction) во время предварительного обучения. В этой задаче модель получает на вход пару предложений. и должна определить, является ли второе предложение продолжением первого. Задача предсказания следующего предложения является задачей бинарной классификации. Благодаря этой задаче сеть обучается различать связь между предложениями в тексте.

Модель BERT может быть обучена на множестве языков и позволяет выполнять обработку текстов на разных языках (в частности, многоязыковая модель BERT-Base поддерживает 104 языка). Это делает BERT гибким инструментом для различных задач, связанных с мультиязычным текстом.

Токенизация в BERT:

• Токенами служат слова, доступные в словаре, или их составные части

• Если слово отсутствует в словаре, оно разбивается на части, которые в словаре присутствуют

В нейронной сети BERT токены кодируются своими эмбеддингами, а именно: соединяются представления самого токена (предобученные), номера его предложения, а также позиции токена внутри своего предложения. Предобученные модели можно подстраивать под решение конкретных NLP-задач. Нужно лишь дообучить модель на данных, специфичных задаче. Так как знание языка уже получено на этапе предобучения языковой модели, необходима лишь коррекция сети.

Пример использования модели BERT в PyTorch:

from transformers import BertModel, BertTokenizer

# Загрузка предобученной модели и токенизатора

model\_name = 'bert-base-uncased'

tokenizer = BertTokenizer.from\_pretrained(model\_name)

model = BertModel.from\_pretrained(model\_name)

# Пример предобработки текста и получения эмбеддингов

text = "Пример текста для обработки"

tokens = tokenizer.tokenize(text)

input\_ids = tokenizer.convert\_tokens\_to\_ids(tokens)

input\_tensor = torch.tensor([input\_ids])

outputs = model(input\_tensor)

embeddings = outputs[0]

1. **Архитектура Transformer.**

Архитектура Transformer — это модель глубокого обучения, представленная в статье "Attention Is All You Need" в 2017 году. Она стала революционным прорывом в области обработки естественного языка и перевода и предложила новый подход к моделированию последовательностей без использования рекуррентных нейронных сетей. Эта архитектура стала основой для многих передовых моделей в областях машинного перевода, обработки естественного языка и других задач. Главной идеей архитектуры Transformer является полное отсутствие рекуррентных и сверточных слоев, которые были широко используемы в предыдущих моделях. Вместо этого она полностью основана на механизме Attention, который позволяет модели эффективно обрабатывать долгосрочные зависимости в последовательностях. Архитектура Transformer состоит из двух основных компонентов: энкодер (Encoder) и декодер (Decoder).

Энкодер:

• Входные данные проходят через несколько блоков энкодера, каждый из которых содержит несколько слоев.

• Каждый слой в блоке энкодера имеет две подсети: Multi-Head Attention (многоголовое внимание) и Feed Forward Network (сеть прямого распространения).

• Многоголовое внимание позволяет моделировать зависимости между различными словами во входной последовательности. Оно обрабатывает входные данные, применяя механизм Attention с несколькими "головами", что позволяет модели фокусироваться на разных аспектах контекста.

• После применения многоголового внимания используется Feed Forward Network для дальнейшей обработки и генерации новых признаков. Позиционно-сетевая сеть применяет полносвязную нейронную сеть к каждой позиции в последовательности независимо, обеспечивая локальное моделирование внутри каждого элемента.

• Процесс проходит через несколько повторяющихся блоков энкодера, что позволяет модели улавливать долгосрочные зависимости и высокоуровневую информацию во входных данных.

Декодер:

• Декодер имеет аналогичную структуру, как энкодер, но с некоторыми дополнительными компонентами.

• На каждом шаге декодер получает входные данные и информацию о контексте из энкодера, чтобы генерировать следующий элемент выходной последовательности.

• Декодер также использует многоголовое внимание, но с дополнительным механизмом маскирования, чтобы обеспечить автономность в генерации и предотвратить доступ к будущим элементам последовательности.

• После многоголового внимания и применения сети прямого распространения, декодер генерирует предсказание для очередного элемента выходной последовательности.

Кроме того, в каждом блоке энкодера и декодера используется метод нормализации слоя и связи-остаточные (резидуальные) соединения (residual connections), чтобы обеспечить более стабильное обучение и легкую передачу информации через слои.

А позиционные эмбеддинги (Positional Embeddings) используются для представления информации о позиции каждого элемента входной последовательности. Они позволяют модели учитывать порядок элементов, не зависимо от их значения.

Преимущества архитектуры Transformer:

*• Параллельная обработка:* все операции внутри энкодера и декодера могут быть выполнены параллельно, что делает Transformer более эффективным для обучения и использования на параллельных архитектурах.

*• Долгосрочные зависимости:* многоголовое внимание позволяет модели улавливать и обрабатывать долгосрочные зависимости в последовательностях, что было проблематично для рекуррентных нейронных сетей.

*• Гибкость и универсальность:* архитектура Transformer может быть применена в различных задачах обработки последовательностей, включая машинный перевод, вопросно-ответные системы, суммаризацию текста и другие, достигая высоких результатов.

Пример использования архитектуры Transformer:

import torch

import torch.nn as nn

from torch.nn import TransformerEncoder, TransformerEncoderLayer

class TransformerModel(nn.Module):

def \_\_init\_\_(self, ntoken, ninp, nhead, nhid, nlayers, dropout=0.5):

super(TransformerModel, self).\_\_init\_\_()

self.encoder = nn.Embedding(ntoken, ninp)

self.pos\_encoder = PositionalEncoding(ninp)

encoder\_layers = TransformerEncoderLayer(ninp, nhead, nhid, dropout)

self.transformer\_encoder = TransformerEncoder(encoder\_layers, nlayers)

self.decoder = nn.Linear(ninp, ntoken)

def forward(self, src):

src = self.encoder(src)

src = self.pos\_encoder(src)

output = self.transformer\_encoder(src)

output = self.decoder(output)

return output

В этом примере мы определяем класс TransformerModel, который представляет собой модель Transformer для обработки последовательностей. Внутри класса мы определяем энкодер, позиционное кодирование, TransformerEncoder и декодер. В методе forward мы применяем последовательно все компоненты архитектуры: кодируем входные данные с помощью энкодера и позиционного кодирования, пропускаем их через TransformerEncoder и проецируем на размерность выходного словаря с помощью линейного слоя.

1. **Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.**

Класс Tensor является основным строительным блоком в библиотеке PyTorch и представляет собой многомерный массив. Он используется для представления данных и выполнения операций в вычислительных графах.

PyTorch предоставляет множество операций для изменения размера тензора. Некоторые из наиболее часто используемых операций включают:

*1. view(\*shape)* - Этот метод позволяет изменить размер тензора, не изменяя его данные. Он принимает новую форму в качестве аргумента shape и возвращает новый тензор с указанной формой.

*2.reshape(\*shape) -* Этот метод также изменяет размер тензора, но может пересчитывать размеры по нескольким осям. Он принимает новую форму в качестве аргумента shape и возвращает новый тензор с указанной формой.

*3.squeeze(dim=None)* - Этот метод удаляет измерения с размером 1 из тензора. Если указан аргумент dim, то он удаляет только измерение с указанным индексом dim. Если аргумент dim не указан, метод удаляет все измерения с размером 1.

*4.unsqueeze(dim)* - Этот метод добавляет новое измерение с размером 1 в тензор. Он принимает аргумент dim, который указывает индекс, на котором нужно добавить новое измерение.

Некоторые из наиболее часто используемых операций агрегации включают:

*• sum(dim=None)* - Вычисляет сумму элементов тензора по указанной размерности dim или по всем измерениям, если dim не указан.

*•* *mean(dim=None)* - Вычисляет среднее значение элементов тензора по указанной размерности dim или по всем измерениям, если dim не указан.

*•* *min(dim=None)* - Находит минимальное значение элементов тензора по указанной размерности dim или по всем измерениям, если dim не указан.

*•* *max(dim=None)* - Находит максимальное значение элементов тензора по указанной размерности dim или по всем измерениям, если dim не указан.

1. **Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.**

Автоматическое дифференцирование является ключевой функцией в библиотеке PyTorch, которая упрощает процесс вычисления градиента функции по отношению к ее входным переменным. Это особенно полезно при обучении нейронных сетей, где требуется оптимизировать параметры модели путем обратного распространения ошибки. Вот как работает автоматическое дифференцирование в PyTorch:

*1. Создание тензоров:* В PyTorch операции выполняются над тензорами, которые представляют многомерные массивы данных. Мы можем создать тензоры и задать параметр `requires\_grad=True`, чтобы указать, что мы хотим вычислять градиенты для этого тензора.

import torch

x = torch.tensor([2.0], requires\_grad=True)

*2. Определение вычислительного графа:* Когда мы выполняем операции с тензорами, PyTorch автоматически создает вычислительный граф, который отслеживает зависимости между операциями.

y = x \*\* 2 + 3 \* x + 1

*3. Вычисление градиентов:* После выполнения всех операций, мы можем вызвать метод `backward()` на финальном тензоре, чтобы автоматически вычислить градиенты для всех тензоров с параметром `requires\_grad=True` в вычислительном графе.

y.backward()

*4. Использование градиентов:* После вызова `backward()`, градиенты будут доступны в атрибуте `grad` каждого тензора.

print(x.grad) # Выводит градиент для x

Пример применения автоматического дифференцирования в обучении нейронной сети:

import torch

import torch.nn as nn

# Определяем нейронную сеть

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.linear = nn.Linear(1, 1)

def forward(self, x):

return self.linear(x)

# Создаем экземпляр модели

model = Net()

# Определяем функцию потерь

loss\_fn = nn.MSELoss()

# Определяем оптимизатор

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Генерируем данные

x = torch.tensor([1.0])

y\_true = torch.tensor([2.0])

# Обучение модели

for epoch in range(100):

# Обнуляем градиенты

optimizer.zero\_grad()

# Прямой проход

y\_pred = model(x)

# Вычисление функции потерь

loss = loss\_fn(y\_pred, y\_true)

# Обратное распространение ошибки

loss.backward()

# Обновление параметров

optimizer.step()

1. **Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).**

Загрузка и преобразование данных являются важной частью процесса обучения моделей машинного обучения. В библиотеке PyTorch для этих целей предоставляются классы Dataset, DataLoader и Transforms.

Класс Dataset представляет собой абстракцию данных и предоставляет методы для доступа к ним. Он позволяет вам определить, какие данные будут загружены и как они будут доступны для обработки. В PyTorch вы можете создать собственный подкласс Dataset, наследуясь от torch.utils.data.Dataset и реализовав методы \_\_len\_\_ (возвращающий размер набора данных) и \_\_getitem\_\_ (возвращающий элемент данных по индексу).

Класс DataLoader обеспечивает эффективную загрузку данных из класса Dataset. Он позволяет вам определить параметры загрузки данных, такие как размер пакета (batch size), перемешивание (shuffle), параллельную загрузку данных и другие. DataLoader предоставляет итератор, который позволяет вам эффективно проходить через данные по эпохам или пакетам.

Transforms в PyTorch представляет собой модуль, который обеспечивает различные преобразования данных, используемых при подготовке и обработке датасетов. Transforms в PyTorch предоставляет удобные средства для преобразования данных перед их использованием в модели. Композиция трансформеров позволяет гибко настраивать последовательность преобразований, что часто используется для предварительной обработки данных и аугментации во время обучения моделей машинного обучения.

Некоторые из часто используемых трансформеров включают в себя:

*• ToTensor():* Преобразует данные в формат тензора. Это особенно полезно при работе с изображениями или аудио, где данные обычно представлены в виде массивов.

*• Normalize(mean, std):* Нормализует данные по заданному среднему значению и стандартному отклонению. Это часто используется для предварительной обработки изображений или числовых данных перед обучением модели.

*• Resize(size):* Изменяет размер изображения на заданный размер. Это полезно для приведения всех изображений в датасете к одному размеру перед обучением модели.

*• RandomCrop(size):* Выполняет случайное обрезание изображения до заданного размера. Это может использоваться для аугментации данных и повышения их разнообразия.

*• RandomHorizontalFlip(p):* Выполняет случайное отражение изображения по горизонтали с заданной вероятностью. Это также может использоваться для аугментации данных.

*• Compose(transforms):* Позволяет объединять несколько трансформеров в цепочку. Все трансформеры в цепочке будут применены последовательно к данным.

Композиция трансформеров в контексте обработки данных означает объединение нескольких преобразований (трансформеров) в цепочку или последовательность для применения их к данным последовательно.

Пример использования Dataset, DataLoader и Transforms:

from torch.utils.data import Dataset, DataLoader

from torchvision.transforms import ToTensor

# Создаем пользовательский класс датасета

class MyDataset(Dataset):

def \_\_init\_\_(self, data):

self.data = data

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, index):

item = self.data[index]

# Применяем преобразование к данным

item = self.transform(item)

return item

def transform(self, item):

# В этом примере используем только одно преобразование - ToTensor

transform = ToTensor()

return transform(item)

# Создаем экземпляр датасета

data = [...] # Ваши данные

dataset = MyDataset(data)

# Создаем загрузчик данных

batch\_size = 32

dataloader = DataLoader(dataset, batch\_size=batch\_size, shuffle=True)

# Итерируем по загрузчику данных

for batch in dataloader:

# Обрабатываем пакет данных

# ...

1. **Класс nn.Module. Назначение. Основные поля, методы.**

Класс nn.Module является основным строительным блоком в библиотеке PyTorch для определения и организации моделей машинного обучения. Он является базовым классом для всех пользовательских моделей в PyTorch и предоставляет ряд полей и методов для удобной работы с моделями.

Назначение класса nn.Module - определение структуры модели: позволяет определить структуру модели, определяя поля (параметры) и методы, которые будут использоваться для выполнения прямого и обратного проходов через модель. Основные поля класса nn.Module:

*• self.training:* Логическое поле, указывающее, находится ли модель в режиме обучения (True) или оценки/инференса (False).

*• self.\_parameters:* Словарь, содержащий все обучаемые параметры модели (например, веса и смещения слоев). Параметры автоматически регистрируются при определении слоев модели с использованием nn.Parameter или других методов, таких как nn.Linear, nn.Conv2d и т. д.

*• self.\_buffers:* Словарь, содержащий временные данные, которые не являются обучаемыми параметрами, но все же могут быть сохранены и использованы внутри модели.

*• self.\_modules:* Словарь, содержащий все подмодули (слои) модели, которые также являются экземплярами nn.Module. Это позволяет создавать иерархические модели, состоящие из нескольких подмодулей.

Основные методы класса nn.Module:

*• \_\_init\_\_():* Метод инициализации класса, в котором определяются все подмодули и поля модели.

*• forward():* Метод, в котором определяется прямой проход через модель. Он должен быть переопределен в пользовательском классе, чтобы указать, как данные проходят через слои модели.

*• parameters():* Метод, возвращающий итератор по обучаемым параметрам модели.

*• named\_parameters():* Метод, возвращающий итератор по обучаемым параметрам модели вместе с их именами.

*• children():* Метод, возвращающий итератор по всем подмодулям модели.

*• to():* Метод, переносящий модель и все ее параметры на указанное устройство (например, GPU).

*• train():* Метод, устанавливающий модель в режим обучения.

*• eval():* Метод, устанавливающий модель в режим оценки/инференса.

1. **Линейные слои (Linear Layers).**

Линейные слои (Linear Layers) — это одни из основных строительных блоков в нейронных сетях, используемых для выполнения линейных преобразований над данными. Они представляют собой полносвязные слои, где каждый входной признак связан со всеми выходными признаками посредством линейного преобразования. В библиотеке PyTorch линейные слои реализованы с использованием класса nn.Linear. Он определяет линейное преобразование, которое можно применить к входным данным, умножая их на матрицу весов и добавляя смещение (bias).

Основные параметры и методы класса nn.Linear:

• nn.Linear(in\_features, out\_features, bias=True):

o in\_features: Число входных признаков (размерность входа).

o out\_features: Число выходных признаков (размерность выхода).

o bias (опционально): Флаг, указывающий, следует ли добавлять смещение (bias) к выходным данным. По умолчанию установлен в True.

• forward(input): Метод, в котором определяется прямой проход через линейный слой. Он принимает входные данные input и возвращает выходные данные, полученные после линейного преобразования. Входные данные input должны быть двумерным тензором с размерностью [batch\_size, in\_features], где batch\_size - размер пакета данных.

Пример создания и применения линейного слоя:

import torch

import torch.nn as nn

# Создаем линейный слой

input\_size = 10

output\_size = 5

linear\_layer = nn.Linear(input\_size, output\_size)

# Создаем входные данные

x = torch.randn(32, input\_size) # Размер пакета 32

# Применяем линейный слой к входным данным

y = linear\_layer(x)

1. **Слои нелинейной активации (Non Linear Activations).**

Слои нелинейной активации (Non-Linear Activation Layers) используются в нейронных сетях для введения нелинейности в модель. Они применяют нелинейные функции к выходу линейного преобразования или других слоев модели, что позволяет сети моделировать более сложные и нелинейные зависимости в данных.

В библиотеке PyTorch нелинейные активации реализованы с использованием различных классов, наиболее распространенные из которых:

*• nn.ReLU:* Функция активации ReLU (Rectified Linear Unit) применяет нелинейное преобразование, оставляя положительные значения без изменений и заменяя отрицательные значения на ноль. Уравнение: ReLU(x) = max(0, x). В PyTorch класс реализуется как nn.ReLU.

*• nn.Sigmoid:* Сигмоидная функция активации преобразует входные значения в диапазон от 0 до 1. Это полезно для задач бинарной классификации или генерации вероятностных оценок. Уравнение: Sigmoid(x) = 1 / (1 + exp(-x)). В PyTorch класс реализуется как nn.Sigmoid.

*• nn.Tanh:* Гиперболический тангенс (tanh) - это нелинейная функция активации, которая преобразует входные значения в диапазон от -1 до 1. Он обладает симметричными свойствами относительно нуля. Уравнение: Tanh(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x)). В PyTorch класс реализуется как nn.Tanh.

*• nn.Softmax:* Функция активации Softmax используется для преобразования вектора значений в вероятностное распределение, где каждое значение на выходе представляет собой вероятность принадлежности к определенному классу. Softmax широко применяется в задачах классификации с множеством классов. В PyTorch класс реализуется как nn.Softmax.

Пример применения нелинейной активации ReLU:

import torch

import torch.nn.functional as F

# Создаем входные данные

x = torch.randn(32, 10) # Размер пакета 32

# Применяем ReLU

y = F.relu(x)

1. **Слои нормализации (Normalization Layers).**

Слои нормализации (Normalization Layers) используются в нейронных сетях для стабилизации и нормализации данных, улучшения скорости обучения и повышения обобщающей способности модели. Они выполняют некоторые преобразования над входными данными или выходами других слоев с целью сделать данные более стабильными и нормализованными. В библиотеке PyTorch существуют различные слои нормализации, наиболее распространенные из которых:

*• nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d:* Слой пакетной нормализации (Batch Normalization) применяется для нормализации входных данных в пределах каждого мини-батча. Он вычисляет среднее и стандартное отклонение для каждого признака по всему мини-батчу и нормализует данные. Это помогает стабилизировать распределение данных и улучшает обобщающую способность модели. В PyTorch классы реализуются как nn.BatchNorm1d для одномерных данных, nn.BatchNorm2d для двумерных данных и nn.BatchNorm3d для трехмерных данных.

*• nn.LayerNorm:* Слой нормализации по слоям (Layer Normalization) выполняет нормализацию данных по всему признаку в пределах каждого примера. Он вычисляет среднее и стандартное отклонение для каждого примера и нормализует данные. В отличие от пакетной нормализации, слой нормализации по слоям не зависит от размера мини-батча и может быть полезным в моделях с переменными размерами входных данных. В PyTorch класс реализуется как nn.LayerNorm.

*• nn.InstanceNorm1d, nn.InstanceNorm2d, nn.InstanceNorm3d:* Слой нормализации по экземплярам (Instance Normalization) выполняет нормализацию данных по каждому примеру в пределах каждого признака. В отличие от пакетной нормализации, слой нормализации по экземплярам не учитывает разные экземпляры в мини-батче и нормализует данные независимо для каждого примера. Это полезно, например, в задачах генерации изображений или стилизации изображений. В PyTorch классы реализуются как nn.InstanceNorm1d для одномерных данных, nn.InstanceNorm2d для двумерных данных и nn.InstanceNorm3d для трехмерных данных.

Пример использования слоя Batch Normalization:

import torch

import torch.nn as nn

# Создаем входные данные

x = torch.randn(32, 10)

# Создаем слой Batch Normalization

bn = nn.BatchNorm1d(10) # 10 - размерность признаков

# Применяем Batch Normalization

y = bn(x)

1. **Класс torch.nn.LSTM и torch.nn.GRU.**

Классы `torch.nn.LSTM` и `torch.nn.GRU` являются реализациями модулей долгой краткосрочной памяти (Long Short-Term Memory, LSTM) и модулей с обновлением ворот (Gated Recurrent Unit, GRU) в фреймворке PyTorch. Они являются часто используемыми модулями для обработки последовательностей данных, таких как тексты, временные ряды и аудио. Оба класса представляют рекуррентные нейронные сети (RNN) и позволяют моделировать долгосрочные зависимости в последовательностях. Они отличаются своей архитектурой и способом обновления внутреннего состояния.

*1. `torch.nn.LSTM`:*

- LSTM (долгая краткосрочная память) является модификацией классической рекуррентной нейронной сети.

- Она имеет механизмы ворот (гейтов) - входного, забывания и выходного гейтов, которые позволяют контролировать поток информации и сохранять или игнорировать информацию в зависимости от ее значимости.

- LSTM хорошо подходит для моделирования зависимостей в длинных последовательностях и предотвращения проблемы затухающего градиента.

- Для использования LSTM в PyTorch, вы можете создать экземпляр класса `torch.nn.LSTM` и передать входные данные через вызов метода `forward`.

*2. `torch.nn.GRU`:*

- GRU (модули с обновлением ворот) является упрощенной версией LSTM.

- Она также имеет ворота, но использует только два вида ворот - входной и обновления.

- GRU обладает меньшим количеством параметров, чем LSTM, и более простой структурой.

- GRU хорошо работает для моделирования зависимостей в последовательностях и является более легковесным альтернативным вариантом LSTM.

- Для использования GRU в PyTorch, вы можете создать экземпляр класса `torch.nn.GRU` и передать входные данные через вызов метода `forward`.

Пример использования `torch.nn.LSTM` и `torch.nn.GRU`:

import torch

import torch.nn as nn

# Пример входных данных

input\_size = 10 # Размерность входных данных

seq\_length = 5 # Длина последовательности

batch\_size = 3 # Размер батча

# Создание LSTM слоя

lstm = nn.LSTM(input\_size=input\_size, hidden\_size=20, num\_layers=2, batch\_first=True)

# Создание GRU слоя

gru = nn.GRU(input\_size=input\_size, hidden\_size=20, num\_layers=2, batch\_first=True)

# Создание случайного тензора входных данных

input\_data = torch.randn(batch\_size, seq\_length, input\_size)

# Применение LSTM к входным данным

lstm\_output, (h\_lstm, c\_lstm) = lstm(input\_data)

# Применение GRU к входным данным

gru\_output, h\_gru = gru(input\_data)

1. **Слои эмбеддингов nn.Embedding и их применение.**

Слои эмбеддингов (Embedding Layers) являются важной составляющей нейронных сетей для обработки текстовых данных. Они предназначены для преобразования дискретных категориальных переменных, таких как индексы слов или идентификаторы категорий, в плотные векторные представления (эмбеддинги), которые можно использовать в качестве входных данных для других слоев нейронных сетей.

Основные параметры класса nn.Embedding включают:

*• num\_embeddings (int):* Общее количество уникальных категорий или слов во входных данных.

*• embedding\_dim (int):* Размерность эмбеддинга, то есть размерность выходного вектора для каждой категории.

*• padding\_idx (int, optional):* Индекс заполнения. Если указан, то соответствующие эмбеддинги будут установлены в нулевой вектор.

Основной метод класса nn.Embedding — это forward, который выполняет преобразование входных индексов в соответствующие эмбеддинги. Входные данные обычно представляют собой индексы слов или категорий, представленные в виде тензора целых чисел.

Некоторые конкретные задачи и сценарии, где применяются слои эмбеддингов:

*• Задачи классификации текста:* В задаче классификации текста, такой как определение тональности отзывов или классификация новостей по тематикам, слои эмбеддингов могут использоваться для представления слов в виде плотных векторов. Это позволяет модели учитывать семантическое значение слов и сходства между ними при принятии решений о классификации.

*• Машинный перевод:* В задаче машинного перевода слои эмбеддингов могут быть использованы для представления слов и предложений на исходном и целевом языках. Это позволяет модели улавливать семантические отношения между словами в разных языках и генерировать более точные переводы.

*• Анализ сентимента:* В задачах анализа сентимента, где требуется определить эмоциональную окраску текста (положительная, отрицательная, нейтральная), слои эмбеддингов могут использоваться для представления слов и последующего обучения моделей классификации. Плотные векторные представления слов позволяют моделям улавливать смысловые оттенки и эмоциональную окраску в тексте.

*• Рекомендательные системы:* В рекомендательных системах, где необходимо предлагать пользователю релевантные товары или контент на основе их предыдущего поведения, слои эмбеддингов могут использоваться для представления товаров, пользователей или контекста. Это позволяет моделям улавливать скрытые зависимости и сходства между товарами и пользователями, улучшая качество рекомендаций.

*• Задача генерации текста:* В задачах генерации текста, таких как автосгенерация текста или генерация подписей к изображениям, слои эмбеддингов могут использоваться для представления слов или символов в тексте. Это позволяет модели генерировать согласованный и информативный текст, учитывая семантическую близость слов и синтаксические правила.

Слои эмбеддингов также могут применяться в других задачах NLP, таких как извлечение информации, вопросно-ответные системы, генерация текста и т. д. Они предоставляют мощный инструмент для представления текстовых данных и извлечения семантических и синтаксических признаков, улучшая производительность моделей в различных задачах обработки текста.

Пример использования слоя эмбеддингов в PyTorch:

import torch

import torch.nn as nn

# Создаем слой эмбеддингов

embedding\_layer = nn.Embedding(num\_embeddings=1000, embedding\_dim=100)

# Создаем пример входных данных

input\_data = torch.LongTensor([[1, 2, 3, 4], [5, 6, 7, 8]])

# Применяем слой эмбеддингов к входным данным

embedded\_output = embedding\_layer(input\_data)

# Выводим размерность полученных эмбеддингов

print(embedded\_output.size())

1. **Слои регуляризации (Dropout Layers).**

Слои регуляризации, такие как Dropout, используются в нейронных сетях для борьбы с переобучением. Переобучение происходит, когда модель слишком точно подстраивается под обучающие данные и теряет способность обобщать на новые данные. Слои регуляризации помогают уменьшить переобучение, снижая сложность модели и улучшая ее обобщающую способность.

Один из наиболее распространенных слоев регуляризации — это Dropout. В Dropout случайным образом выбираются некоторые нейроны и их выходы обнуляются (отключаются) во время обучения с некоторой вероятностью. Это означает, что в каждой эпохе обучения некоторые нейроны игнорируются, и модель вынуждена учиться на основе оставшихся активных нейронов. В результате Dropout принудительно делает модель более устойчивой и предотвращает слишком сильную адаптацию к конкретным обучающим примерам. Применение Dropout слоя помогает снизить переобучение и улучшить обобщающую способность модели. Он также может способствовать обучению более устойчивых и универсальных признаков. В PyTorch слой Dropout представлен классом `torch.nn.Dropout`. Пример использования Dropout:

import torch

import torch.nn as nn

# Создаем входные данные

x = torch.randn(32, 10)

# Создаем слой Dropout с вероятностью отключения 0.5

dropout = nn.Dropout(p=0.5)

# Применяем Dropout

y = dropout(x)

1. **Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).**

Сверточные слои (Convolution Layers) и сжимающие слои (Pooling Layers) являются основными компонентами сверточных нейронных сетей, которые широко применяются в задачах обработки изображений и анализа видео.

Сверточные слои применяют операцию свертки к входным данным, чтобы выделить локальные пространственные шаблоны и признаки. Они состоят из набора фильтров (ядер свертки), которые применяются к входным данным, скользящим окном с заданным шагом. Каждый фильтр выделяет определенные признаки, такие как границы, текстуры или формы, в зависимости от обучения. Сверточные слои имеют общие веса, что позволяет им распознавать признаки в разных частях изображения.

Сжимающие слои, также известные как слои пулинга (Pooling Layers), применяются после сверточных слоев для уменьшения размерности пространственных карт признаков. Они выполняют операцию агрегации, которая объединяет близлежащие признаки в одно значение, сохраняя наиболее важные характеристики. Наиболее распространенная операция пулинга — это операция максимального пулинга (Max Pooling), которая выбирает максимальное значение из каждой области пулинга.

Сверточные слои и сжимающие слои позволяют сверточным нейронным сетям эффективно обрабатывать изображения, сохраняя пространственную структуру и локальные зависимости. Они уменьшают количество параметров и вычислительную сложность модели, что позволяет обучать глубокие нейронные сети на больших наборах данных.

Примеры сверточных слоев и слоев пулинга в PyTorch:

import torch

import torch.nn as nn

# Создаем входные данные размером (батч, каналы, высота, ширина)

x = torch.randn(32, 3, 64, 64)

# Применяем сверточный слой с 16 фильтрами и размером ядра 3x3

conv = nn.Conv2d(3, 16, kernel\_size=3)

# Применяем слой пулинга с размером окна 2x2

pool = nn.Max

Pool2d(2)

# Применяем сверточный слой

y\_conv = conv(x)

# Применяем слой пулинга

y\_pool = pool(y\_conv)

1. **Слои функций потерь (Loss Functions).**

Слои функций потерь (Loss Functions) являются ключевыми компонентами обучения нейронных сетей. Они определяют, насколько хорошо модель выполняет свою задачу и предоставляют метрику для оценки разницы между предсказанными значениями модели и фактическими значениями.

В библиотеке PyTorch функции потерь реализованы с использованием класса nn.Module и его подклассов. Основная функция потерь, такая как кросс-энтропия или среднеквадратичная ошибка, обычно представлена в виде класса, который наследуется от nn.Module.

Основные методы и свойства классов функций потерь:

*• forward(input, target):* Метод, который вычисляет значение функции потерь на основе входных данных input и целевых значений target. Он возвращает значение функции потерь.

*• \_\_call\_\_(input, target):* Функция вызова, которая делегирует выполнение операции forward. Позволяет вызывать объект функции потерь напрямую, как функцию, передавая ей входные данные и целевые значения.

Некоторые распространенные функции потерь в PyTorch:

*• nn.MSELoss:* Среднеквадратичная ошибка (Mean Squared Error). Вычисляет среднее значение квадратов разностей между предсказанными и целевыми значениями.

*• nn.CrossEntropyLoss:* Кросс-энтропия. Часто используется для задач классификации с несколькими классами. Принимает на вход логиты (не нормализованные значения) и целевые метки классов.

*• nn.BCELoss:* Бинарная кросс-энтропия. Используется для задач бинарной классификации. Принимает на вход логиты и бинарные целевые метки.

*• nn.NLLLoss:* Отрицательный логарифм правдоподобия (Negative Log Likelihood). Используется для задач классификации, когда входные данные уже подверглись логарифмическому преобразованию, например, с помощью nn.LogSoftmax или nn.LogSigmoid.

Это только некоторые примеры функций потерь в PyTorch. Библиотека также предоставляет множество других функций потерь, и в некоторых случаях вы можете определить свою собственную функцию потерь, наследуясь от класса nn.Module.

Пример использования функции потерь в PyTorch:

import torch

import torch.nn as nn

# Создаем пример предсказаний и истинных меток

predictions = torch.randn(64, 10) # предсказанные значения

labels = torch.randint(0, 10, (64,)) # истинные метки классов

# Используем функцию потерь CrossEntropyLoss

loss\_fn = nn.CrossEntropyLoss()

# Вычисляем значение функции потерь

loss = loss\_fn(predictions, labels)

# Выводим значение потери

print(loss.item())

1. Модель перцептрона. Проблема линейно неразделимых множеств и ее решение. Логика построения многослойных ИНС.
2. Линейное отображение. Векторно-матричное дифференцирование.
3. Функции активации. Требования к функциям активации Популярные функции активации.
4. Адаптивные методы градиентного спуска. Метод импульсов. Метод Нестерова.
5. Глубокое обучение. «Вторая весна искусственного интеллекта» и ее причины.
6. Преобразование Softmax и функция потерь Cross Entropy loss.
7. Механизм обратного распространения ошибки.
8. Кросс-валидация. Выборки train, validation, test. Проблема переобучения. Ранняя остановка.
9. Проблема поиска градиента в общей логике обучения нейронной сети. Градиент функции многих переменных. Методы вычисления.
10. Дифференцируемое программирование и реализация обратного распространения ошибки.
11. Стохастический градиентный спуск. Батчи обучающей выборки.
12. Проблема инициализации весов при обучении ИНС. Инициализация Ксавье.
13. Гиперпараметры. Скорость обучения и размер батча.
14. Переобучение модели и регуляризация. Dropout.
15. Минбатчи – причина использования. Нормализация по мини-батчам.
16. Специфика задач машинного обучения на изображениях. Принцип работы сверточных сетей. Преимущества сверточных сетей при решении этих задач.
17. Архитектура многослойной ИНС распознавания изображений на основе сверточных сетей.
18. Принципиальная логика обучения нейронной сети.
19. Приемы для глубокого обучения на небольших наборах изображений.
20. Схема работ слоя сверточной сети. Пулинг. Гиперпараметры: padding, kernel size, stride, dilation.
21. Задачи обработки текста: дистрибутивная семантика, матрица совместной встречаемости, представление слов в виде векторов малой размерности.
22. Word2vec: модель CBOW.
23. Многослойные сети. Граф потока вычислений.
24. Word2vec: модель Skip-Gram.
25. Механизм Attention. Пример использования Attentinon.
26. Рекуррентная нейронная сеть, принципы ее обучения. Сложности применения рекуррентных нейронных сетей.
27. Модуль LSTM.
28. Модель BERT.
29. Архитектура Transformer.
30. Класс Tensor. Операции, изменяющие размер тензора. Операции агрегации.
31. Автоматическое дифференцирование в PyTorch. Пример и применение в обучении ИНС.
32. Загрузка и преобразование данных. Классы Dataset, DataLoader, Transforms (и композиция трансформеров).
33. Класс nn.Module. Назначение. Основные поля, методы.
34. Линейные слои (Linear Layers).
35. Слои нелинейной активации (Non Linear Activations).
36. Слои нормализации (Normalization Layers).
37. Класс torch.nn.LSTM и torch.nn.GRU.
38. Слои эмбеддингов nn.Embedding и их применение.
39. Слои регуляризации (Dropout Layers).
40. Сверточные слои (Convolution Layers). Сжимающие слои (Pooling Layers).
41. Слои функций потерь (Loss Functions).