Алгоритмы оптимизации нейронной сети

Стохастический градиентный спуск (SGD)

$$g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

$$\theta = \theta - \epsilon_k \times g$$

Стандартное накопление импульса

$$v = \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta), y^{(i)}) \right)$$

$$\theta = \theta + v$$

Импульс Нестерова

$$v = \alpha v - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta + \alpha \times v), y^{(i)}) \right)$$

$$\theta = \theta + v$$

AdaGrad

$$g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

$$s = s + g^T g$$

$$\theta = \theta - \epsilon_k \times g / \sqrt{s + eps}$$

RMSProp

$$g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

$$s = \text{decay\_rate} \times s + (1 - \text{decay\_rate}) g^T g$$

$$\theta = \theta - \epsilon_k \times g / \sqrt{s + eps}$$

Адам

$$g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$$

$$m = \beta_1 m + (1 - \beta_1) g$$

$$s = \beta_2 s + (1 - \beta_2) g^T g$$

$$\theta = \theta - \epsilon_k \times m / \sqrt{s + eps}$$

Подготовка набора данных

```
datagen = ImageDataGenerator(rescale=1. / 255)
```

Формирование обучающей выборки

```
train_generator = datagen.flow_from_directory(
 train_dir,
 target_size=(img_width, img_height),
 batch_size=batch_size,
 class_mode='categorical')
```

Формирование валидационной выборки.

```
val_generator = datagen.flow_from_directory(
    val_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical')
```

Формирование тестовой выборки

```
test_generator = datagen.flow_from_directory(
    test_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical')
```

Компиляция

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

Обучение

```
model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=epochs,
    validation_data=val_generator,
    validation_steps=nb_validation_samples // batch_size)
```

Перекрёстная энтропия

Расстояние Кульбака – Лейблера или относительная энтропия

$$\mathrm{KL}(P\|Q) = \int \log \frac{\mathrm{d}P}{\mathrm{d}Q} \mathrm{d}P,$$

- когда P и Q — дискретные случайные величины на дискретном множестве $X = \{x_1, \ldots, x_N\}$, расстояние Кульбака – Лейблера выглядит так:

$$\mathrm{KL}(P\|Q) = \sum_i p(x_i) \log \frac{p(x_i)}{q(x_i)},$$

  где $p(x_i)$ и $q(x_i)$ — собственно вероятности исхода x_i;
- когда P и Q — непрерывные случайные величины в пространстве $\mathbb{R}^d$, расстояние Кульбака – Лейблера можно записать как

$$\mathrm{KL}(P\|Q) = \int_{\mathbb{R}^d} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})} \mathrm{d}\boldsymbol{x},$$

  где p и q — плотности распределений p и q.

Cross - entropy

$$H(p, q) = \mathbb{E}_p [-\log q] = -\sum_y p(y) \log q(y).$$

Связь перекрёстной энтропии и расстояния Кульбака - Лейблера

$$\mathrm{KL}\,(P\|Q) = \sum_y p(y) \log \frac{p(y)}{q(y)} =$$

$$= \sum_y p(y) \log p(y) - \sum_y p(y) \log q(y) = H(p) + H(p,q),$$

Целевая функция для бинарной классификации

Для бинарной классификации целевая функция, которую мы будем минимизировать на наборе данных $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, обычно выглядит как средняя перекрестная энтропия по всем точкам в данных:

$$L(\boldsymbol{\theta}) = H(p_{\text{data}}, q(\boldsymbol{\theta})) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log \hat{y}_i(\boldsymbol{\theta}) + (1 - y_i)\log\left(1 - \hat{y}_i(\boldsymbol{\theta})\right)\right),$$

где $\hat{y}_i(\boldsymbol{\theta})$ — оценка вероятности ответа 1, полученная классификатором.

Формула Байеса для двух классов

$$p(C_1 \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid C_1)p(C_1)}{p(\boldsymbol{x} \mid C_1)p(C_1) + p(\boldsymbol{x} \mid C_2)p(C_2)}.$$

Перепишем это равенство чуть по-другому:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)} = \frac{1}{1 + e^{-a}} = \sigma(a),$$

где

$$a = \ln \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)}, \qquad \sigma(a) = \frac{1}{1 + e^{-a}}.$$

Для входного набора данных $\{\boldsymbol{x}_n, t_n\}$, где $\boldsymbol{x}_n$ — входы, а t_n — соответствующие им правильные ответы, $t_n \in \{0,1\}$, мы получаем такое правдоподобие:

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{1-t_n}, \quad \text{где} \quad y_n = p(C_1 \mid \boldsymbol{x}_n).$$

И теперь можно искать параметры максимального правдоподобия, максимизируя $\ln p(\boldsymbol{t} \mid \boldsymbol{w})$, то есть минимизируя следующую функцию:

$$E(\boldsymbol{w}) = -\ln p(\boldsymbol{t} \mid \boldsymbol{w}) = -\sum_{n=1}^{N} \left[t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right].$$

k классов

$$p(C_k \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid C_k)p(C_k)}{\sum_{j=1}^{K} p(\boldsymbol{x} \mid C_j)p(C_j)} = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}.$$

Теперь у нас столько же аргументов, сколько классов: $a_k = \ln p(\boldsymbol{x} \mid C_k)p(C_k)$. И оптимизируем мы все ту же функцию правдоподобия, что и для двух классов. Давайте закодируем поступающие на вход правильные ответы в виде векторов длины K, в каждом из которых все компоненты равны нулю, кроме правильного класса, где стоит единица (это называется *one-hot* кодированием, и мы с ним не раз еще встретимся). Тогда для таких векторов $\boldsymbol{T} = \{\boldsymbol{t}_n\}$ правдоподобие выглядит так:

$$p(\boldsymbol{T} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k \mid \boldsymbol{x}_n)^{t_{nk}} = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}},$$

где $y_{nk} = y_k(\boldsymbol{x}_n)$. И снова можно взять производную и прийти к тому же самому выражению, только в сумме станет больше слагаемых:

$$E(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = -\ln p(\boldsymbol{T} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln y_{nk}.$$

Сохранение обученной нейронной сети

- Обучение нейронной сети требует длительного времени
- Рекомендуется сохранять сеть для последующего использования

Полное сохранение нейронной сети

- Архитектура, обученные веса, конфигурация обучения, состояние оптимизатора

Сохранение отдельных компонентов сети

- Сохранение архитектуры сети
- Сохранение весов сети

Сохранение обученной нейронной сети

Формат файла для записи нейронной сети

- HDF5 (Hierarchical Data Format v5)
- https://www.hdfgroup.org/solutions/hdf5/

Что сохраняется при записи нейронной сети в файл:

- Архитектура нейронной сети
- Веса обученной нейронной сети
- Конфигурация обучения (функция ошибки, тип оптимизатора)
- Состояние оптимизатора

Загрузка сохраненной нейронной сети

```
from tensorflow.keras.models import load_model
# Загрузка сети из файла
model = load_model('fashion_mnist_dense.h5’)
# Применение сети для распознавания объектов
prediction = model.predict(x_test)
```

Что происходит при загрузке сети

Создается модель Keras с описанной в файле архитектурой

В модель загружаются веса из файла

Выполняется компиляция модели на основе конфигурации обучения в файле

Сохранение и загрузка весов модели

```
# Сохранение весов модели
model.save_weights('fashion_mnist_weights.h5')
# Загрузка весов в модель
model.load_weights('fashion_mnist_weights.h5')
```

Сохранение архитектуры сети

```
# Получение архитектуры сети в формате JSON
json_string = model.to_json()
# Загрузка архитектуры сети из формата JSON
from tensorflow.keras.models import model_from_json
model = model_from_json(json_string)
```