Вопросы hard

1. Линейный слой – слой, в котором выполняется линейное преобразование входных данных. Формула:

$$y = Wx + b,$$

где x – вектор входных данных, y – вектор выходных данных, W – матрица параметров, b – вектор смещений.

Универсальная теорема аппроксимации (теорема Цыбенко): с помощью нейросети с одним скрытым слоем и произвольным количеством нейронов (в качестве функции активации используется сигмоида) можно аппроксимировать любую непрерывную функцию. Нюанс: для удовлетворительной аппроксимации может потребоваться большое число нейронов.

Нелинейность в нейросетях позволяет улавливать сложные зависимости в данных (в противном случае любая нейросеть представляла бы простую линейную комбинацию вне зависимости от числа слоев и нейронов). Пример:

а) AND: $y = \sigma(x_1 + x_2 - 1.5)$, где σ – пороговая функция активации;

б) OR: $y = \sigma(x_1 + x_2 - 0.5)$;

в) XOR: $y = \sigma(\sigma(-x_1 - x_2 + 1) + \sigma(x_1 + x_2 - 0.5) - 1.5)$;

Функции активации (нелинейности):

а) сигмоида: $\sigma(x) = \frac{1}{1+e^{-x}}$. Достоинства: возвращает значения в диапазоне [0; 1] (представляет собой теоретическую вероятность). Недостатки: несимметрична относительно 0 (смещает градиенты); в глубоких нейросетях градиенты затухают;

б) гиперболический тангенс: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. Достоинства: возвращает значения в диапазоне [-1; 1] (градиенты не смещены). Недостатки: затухание градиентов (как у сигмоиды);

в) ReLU: $ReLU(x) = \begin{cases} 0, x \leq 0, \\ x, x > 0. \end{cases}$ Достоинства: нет затухания градиентов при x > 0. Недостатки: затухание градиентов при x <= 0 сохраняется; несимметрична относительно 0 (смещает градиенты); не дифференцируема в 0;

г) Leaky ReLU: $LeakyReLU(x) = \begin{cases} ax, x \leq 0, \\ x, x > 0. \end{cases}$ Достоинства: решается проблема затухания градиентов при x <= 0. Недостатки: требуется подбор параметра a.

Количество параметров в линейном слое нейросети можно посчитать как $(1 + \text{количество входов}) * \text{количество выходов}$.

Backpropagation – способ расчета градиента сложной функции «с конца»:

$$f(x) = g_m(g_{m-1}(\dots(g_1(x))\dots)) \to \frac{df}{dx} = \frac{dg_m}{dg_{m-1}} \cdot \dots \cdot \frac{dg_1}{dx}.$$

Пример для одного слоя, лосс – MSE, w1 = 1, w2 = 2, target = 1. Прямой проход:

$$h = 0.5w_1 + 0.2w_2 - 1.1 = 0.5 + 0.4 - 1.1 = -0.2,$$

$$\sigma = \sigma(h) = \sigma(-0.2) = 0.45,$$

$$MSE = \frac{(target - \sigma)^2}{2} = \frac{(1 - 0.45)^2}{2} = 0.15.$$

Обратный проход для w1:

$$\frac{dMSE}{d\sigma} = -(target - \sigma) = -(1 - 0.45) = -0.55,$$

$$\frac{d\sigma}{dh} = \sigma(h)\big(1 - \sigma(h)\big) = 0.2025,$$

$$\frac{dh}{dw_1} = 0.5 \to$$

$$\frac{dMSE}{dw_1} = \frac{dMSE}{d\sigma} \cdot \frac{d\sigma}{dh} \cdot \frac{dh}{dw_1} = -0.55 \cdot 0.2025 \cdot 0.5 = 0.0557.$$

2. Виды градиентного спуска:

а) GD – расчет производим по всем объектам в выборке:

$$w_{new} = w_{old} - \eta \bigtriangledown Q(w_{old}).$$

Достоинства: точность расчета. Недостатки: вычислительно затратный метод (надо рассчитать $l * d$ производных, где l – количество объектов, d – количество признаков).

б) SGD – расчет выполняем для одного объекта:

$$w_{new} = w_{old} - \eta \bigtriangledown q_{ind}(w_{old}).$$

Достоинства: быстрота расчета. Недостатки: плохая сходимость из-за низкой точности.

в) mini-batch GD – делим датасет на батчи и расчет производим для набора батчей:

$$w_{new} = w_{old} - \frac{\eta}{N}\sum \nabla\, q_i(w_{old}).$$

Компромисс между скоростью и точностью расчета.

Модификации градиентного спуска:

а) метод моментов (позволяет выходить из локальных минимумов):

$$w_t = w_{t-1} - h_t,$$

$$h_t = ah_{t-1} + \eta \nabla Q(w_{t-1}),$$

$$h_t = ah_{t-1} + \eta \nabla Q(w_{t-1} - ah_{t-1}) \text{ (момент Нестерова)}.$$

б) AdaGrad (адаптивный шаг обучения: чем больше двигались на предыдущих шагах, тем меньше будем двигаться на текущем):

$$G_{j,t} = G_{j,t-1} + g_{j,t}^2,$$

$$w_{j,t} = w_{j,t-1} - \frac{\eta}{\sqrt{G_{j,t} + \varepsilon}} g_{j,t}.$$

в) RMSProp (решает проблему монотонного затухания AdaGrad):

$$G_{j,t} = aG_{j,t-1} + (1-a)g_{j,t}^2,$$

$$w_{j,t} = w_{j,t-1} - \frac{\eta}{\sqrt{G_{j,t} + \varepsilon}} g_{j,t}.$$

г) Adam (метод моментов + адаптивный шаг; при движении без минимумов получаем низкую дисперсию спуска, при попадании в минимум – высокую):

$$w_{j,t} = w_{j,t-1} - \frac{\eta}{\sqrt{\hat{G}_{j,t} + \varepsilon}} \hat{h}_{j,t}.$$

$$h_{j,t} = \beta_1 h_{j,t-1} + (1-\beta_1)g_{j,t}, \hat{h}_{j,t} = \frac{h_{j,t}}{1-\beta_1^t}.$$

$$G_{j,t} = \beta_2 G_{j,t-1} + (1-\beta_2)g_{j,t}^2, \hat{G}_{j,t} = \frac{G_{j,t}}{1-\beta_2^t}.$$

д) AdamW (Adam + регуляризация; регуляризация весов отделяется от градиентного спуска и действует одинаково для всех параметров):

$$w_{j,t} = w_{j,t-1} - \eta\left(\frac{1}{\sqrt{\hat{G}_{j,t} + \varepsilon}}\hat{h}_{j,t} + \lambda w_{j,t-1}\right).$$

Лоссы для задачи классификации:

а) бинарная кросс-энтропия:

$$L = -\frac{1}{N}\sum\left(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right).$$

б) мультиклассовая кросс-энтропия:

$$L = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{C}\left(y_{ij}\log(p_{ij})\right).$$

Достоинства: связана с задачей максимизации правдоподобия. Недостатки: чувствительность к дисбалансу классов.

Лоссы для задачи регрессии:

а) MSE:

$$L = \frac{1}{N}\sum(y_i - \hat{y}_i)^2.$$

Достоинства: гладкая и дифференцируемая. Недостатки: чувствительна к выбросам. Недостатки: чувствительна к выбросам; постоянные градиенты ±1.

б) MAE:

$$L = \frac{1}{N}\sum|y_i - \hat{y}_i|.$$

Достоинства: нечувствительна к выбросам. Недостатки: не дифференцируема в 0.

в) RMSE:

$$L = \sqrt{\frac{1}{N}\sum(y_i - \hat{y}_i)^2}.$$

Единицы измерения такие же, как в MAE. Градиенты непостоянны, эта функция тоже чувствительна к выбросам.

Как влияет выбор lr: при слишком больших lr функция не будет сходиться (не попадет в минимум); при слишком малых lr модель будет слишком медленно сходиться и может попасть в локальный минимум. Имеет смысл уменьшать шаг обучения при схождении в минимум. Также для некоторых архитектур имеет смысл увеличивать шаг в начале обучения.

3. Правильная инициализация весов позволяет избежать таких проблем, как взрыв или затухание градиента уже при старте обучения.

Основные виды инициализации:

Xavier Glorot

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

Kaiming He

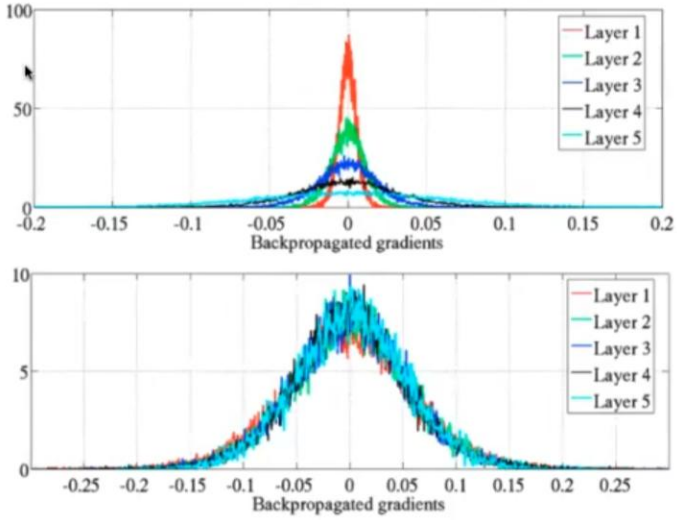
$$W \sim \mathcal{N}\left(0, \frac{2}{n^l}\right)$$

https://paperswithcode.com/paper/delving-deep-into-rectifiers-surpassing-human

для инициализации Ксавьера nj – входная размерность, n(j+1) – выходная размерность.

Инициализация Ксавьера предназначена для гиперболического тангенса и сигмоиды. Инициализация Хе предназначена для ReLU.

Пример для инициализации Ксавьера: верхняя картинка описывает ситуацию без инициализации, нижняя – с инициализацией:



Почему нельзя использовать константу при инициализации весов: в таком случае каждый нейрон будет считать одинаковые выходы и выучивать одни и те же параметры.

4. Dropout - это метод регуляризации, который применяется в нейронных сетях для предотвращения переобучения. Он заключается в случайном отбрасывании некоторых нейронов в процессе обучения. Это означает, что во время каждой итерации обучения выходы случайно выбранных нейронов устанавливаются в ноль. Процесс отбрасывания происходит независимо для каждого нейрона. Dropout может применяться на всех слоях, кроме выходного.

На этапе train отключаем нейроны с фиксированной вероятностью p по формуле:

$$y = f(Wx) \cdot m,\ m \sim Bernoulli(1-p).$$

На этапе eval имитируем вероятность присутствия нейрона:

$$y = (1-p)f(Wx).$$

Зачастую удобнее никак не менять процесс применения обученной сети, поэтому стандартный Dropout обучают по формуле:

$$y = \frac{1}{1-p} f(Wx) \cdot m,\ m \sim Bernoulli(1-p).$$

Тогда на этапе eval домножать на 1−p не потребуется.

Процедура BatchNorm позволяет избавиться от влияния обучения предыдущих слоев нейросети на обучение последующих. Поясним:

а) Сигнал, который идет от нейрона при обучении на различных батчах изначально имеет нормальное распределение с некоторым средним значением.

б) При обучении мы ожидаем, что сигнал, идущий в и из каждого нейрона, имеет некоторое симметричное (допустим, нормальное) распределение.

в) Но так как изменение веса на одном слое (при обучении сети) влияет на изменение весов на соседних слоях, то распределение сигнала меняется - может измениться среднее значение, а также поменяется разброс.

Данное обстоятельство замедляет обучение нейросети и снижает качество.

Идея слоя BatchNorm: пусть дан мини-батч. Сначала вычисляем среднее значение по батчу:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Затем вычисляем дисперсию по батчу:

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

Теперь центрируем значения батча и нормируем дисперсию - это Batch Normalization:

$$x_i^{new} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Тем самым, мы приводим распределение значений нейрона к некоторому фиксированному среднему и дисперсии, и оно становится контролируемым. Однако не факт, что у каждого нейрона распределение должно иметь нулевое среднее и единичную дисперсию, поэтому затем итоговое значение нейрона в слое BatchNorm вычисляем по формуле:

$$y_i = \gamma x_i^{new} + \beta,$$

где β, γ – обучаемые параметры.

На этапе train: в pyTorch по умолчанию считается скользящее среднее (считаем средние и дисперсии по батчам, их усредняем) с применением momentum.

На этапе eval: среднее и дисперсия либо используются те же, что и на этапе обучения, либо (при флаге track_running_stats=False) считаются по батчу, что прилетел на тесте.

Достоинства BatchNorm:

а) Делает обучение более стабильным и увеличивает его скорость.

б) Иногда позволяет убрать Dropout.

в) Из-за ускорения обучения позволяет использовать более глубокие сети.

г) Уменьшает чувствительность к инициализации весов.

Нормализация данных почти всегда критически важна для нейронных сетей, и это применимо ко всем архитектурам, включая как полносвязные сети, так и сверточные сети.

5. Формула сверточного слоя:

$$\mathrm{Im}^{\mathrm{out}}(x, y, t) = \sum_{i=-d}^{d} \sum_{j=-d}^{d} \sum_{c=1}^{c} \left(K_t(i, j, c)\, \mathrm{Im}^{in}(x + i, y + j, c) + b_t \right)$$

Количество параметров в сверточном слое:

Ответ: $K \cdot K \cdot C_{in} \cdot C_{out} + C_{out}$, где K - размер ядра, C - количество каналов.

Количество операций умножения в свертке:

Ответ: $H \cdot W \cdot C_{out} \cdot (K \cdot K \cdot C_{in})$, где K - размер ядра, C - количество каналов, H и W - высота и ширина выходного тензора.

Параметры сверточного слоя:

а) padding – задает рамку вокруг изображения (рамка может быть заполнена нулями, может быть зеркальным отображением картинки и т.п.);

б) dilation – разреженность фильтра;

в) stride – шаг фильтра.

Размер выходного изображения с учетом всех параметров сверточного слоя:

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \mathrm{padding}[0] - \mathrm{dilation}[0] \times (\mathrm{kernel\_size}[0] - 1) - 1}{\mathrm{stride}[0]} + 1 \right\rfloor$$

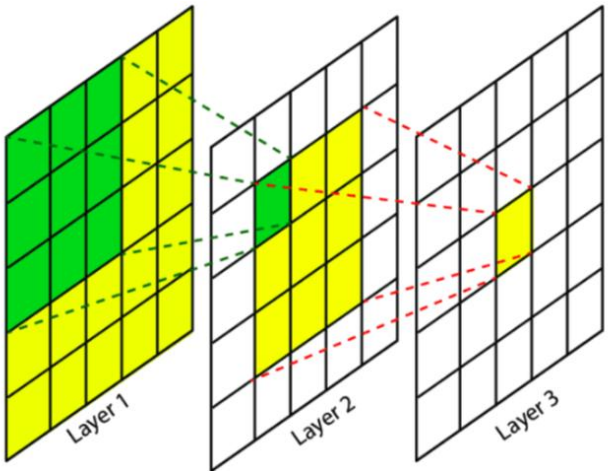
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \mathrm{padding}[1] - \mathrm{dilation}[1] \times (\mathrm{kernel\_size}[1] - 1) - 1}{\mathrm{stride}[1]} + 1 \right\rfloor$$

Размер выходного изображения с учетом только паддинга:

Ответ: $O = \frac{W - F + 2P}{S} + 1$, где W — размер входа, F — размер ядра, P — паддинг, S — шаг.

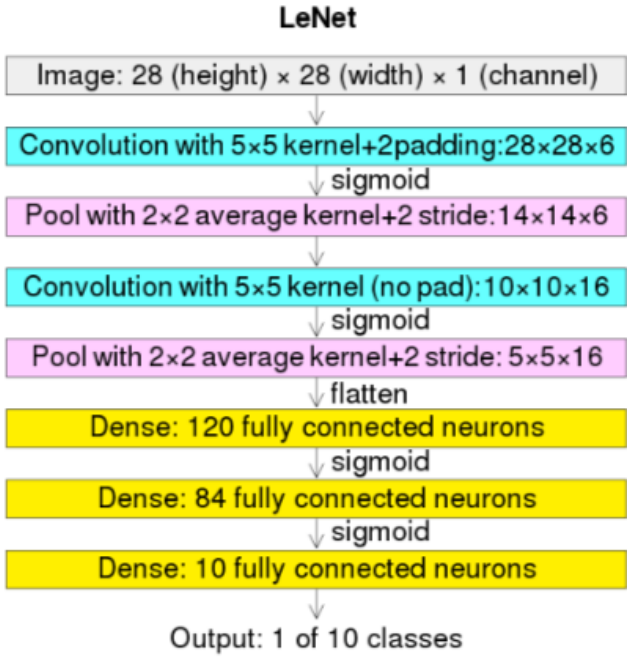
Поле восприятия (receptive field) – это то, сколько пикселей из исходного изображения видит пиксель после определенного слоя. Например, если сеть - фильтр 3x3, то RF выходного пикселя - 3x3, потому что он получился из суммирования произведений фильтра 3x3 с областью исходного изображения размера 3x3. Если сеть содержит 2 слоя, то пиксель с последнего слоя видит

3x3 пикселей со второго слоя. Каждый пиксель со второго слоя, в свою очередь, видит 3x3 пикселей изначального изображения. При перемещении фильтров часть пикселей будет пересекаться, поэтому, как видно на картинке, поле восприятия получается 5x5.



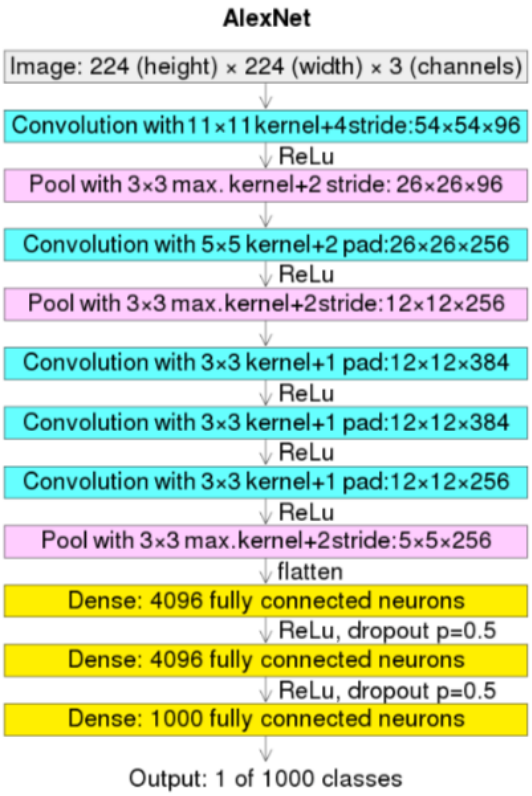
6. Основные архитектуры CNN:

а) LeNet – первая CNN:



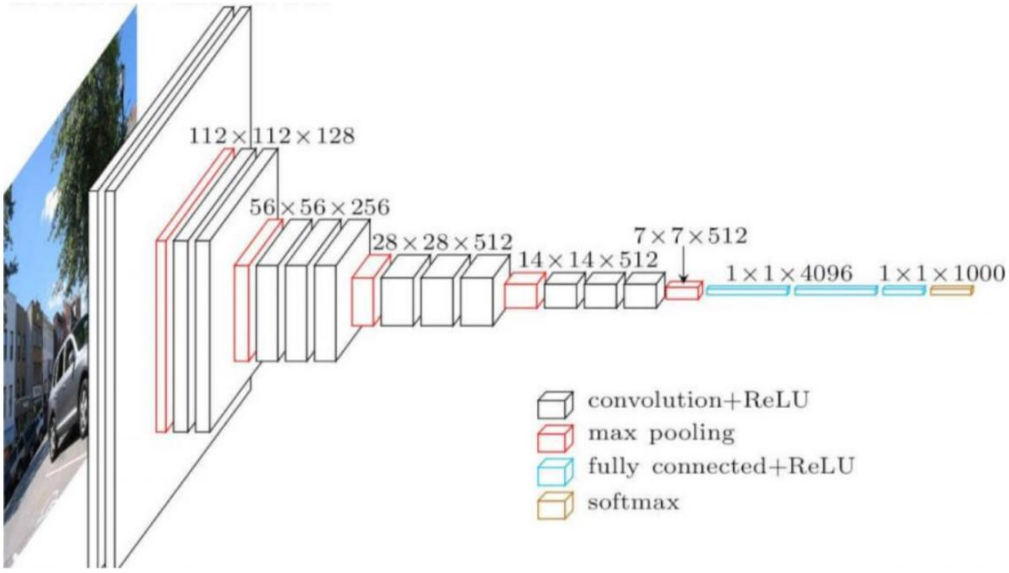
Недостатки: простая устаревшая архитектура.

б) AlexNet – вторая по старшинству нейросеть:



Недостатки: лучше LeNet, но все равно устаревшая архитектура.

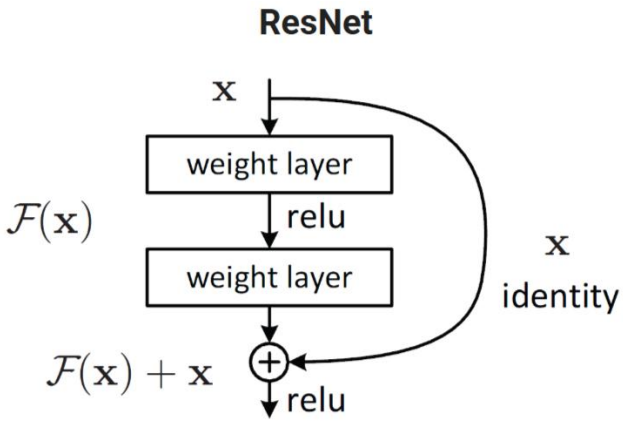
в) VGG:



Использует небольшие свертки (3x3), но имеет большую глубину. Недостатки: большой вес параметров (0.5 Гб) и долгое обучение.

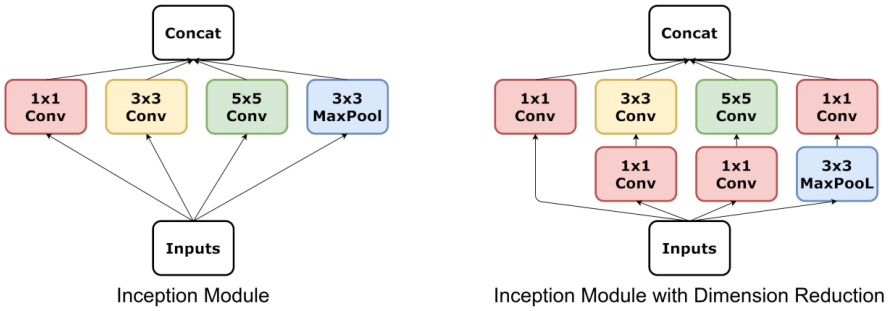
г) ResNet – архитектура с использованием residual connections. Нейросеть учит остаток между входом X и требуемым выходом Y:

$$F(X) = Y - X$$



Достоинства: позволяет избежать затухания градиентов и передать больше информации с предыдущих слоев. Недостатки: большая глубина сетей.

д) Inception – использует разделение на параллельные операции:



Достоинства: сокращение числа параметров (вычислительная эффективность), параллельное обучение позволяет извлечь разные признаки. Недостатки: сложность расчета градиентов.

Сейчас имеет смысл использовать ResNet, Inception.

7. Transfer Learning — это техника машинного обучения, при которой знания, полученные моделью при решении одной задачи, используются в качестве отправной точки для решения новой, но связанной задачи. Вместо обучения модели "с нуля", мы берем предобученную модель (обычно на большом и общем датасете) и адаптируем ее под свою конкретную задачу с меньшим объемом данных.

Преимущества:

а) Экономия ресурсов.

б) Облегчение работы с малыми данными.

в) Улучшение производительности.

Основные подходы к transfer learning:

а) Feature Extraction. Суть: предобученная модель используется как "экстрактор фич". Удаляются ее последние полностью связанные слои (которые отвечают за классификацию на исходные классы), и на выход оставшейся "базовой" модели добавляется новый классификатор (обычно один или несколько полносвязных слоев + выходной слой с числом нейронов, равным числу классов новой задачи). Обучение: "Замороженные" веса базовой модели не обновляются во время обучения. Обучаются только веса нового классификатора. Когда использовать: когда новый датасет очень мал или очень похож на исходный датасет.

б) Fine-Tuning. Суть: мы не только заменяем и обучаем новый классификатор, но и частично "размораживаем" и дообучаем некоторые слои базовой предобученной модели. Процесс: заменить и обучить новый классификатор на замороженной базе; «разморозить» один или несколько верхних (ближе к выходу) сверточных блоков базы; продолжить обучение всех размороженных слоев и нового классификатора, но с очень маленькой скоростью обучения (learning rate), чтобы не "испортить" уже полезные признаки слишком большими шагами обновления весов. Когда использовать: когда новый датасет достаточно большой и/или когда новая задача специфична и отличается от исходной.

Пример Fine-Tuning: классификация кошек и собак (2 класса) с помощью нейросети, обученной на датасете ImageNet (1000 классов).

"Заморозить" веса слоя означает исключить их из списка параметров, обновляемых в процессе backpropagation и оптимизации (в PyTorch задается как param.requires_grad = False). Заморозка позволяет сохранить качество нейросети.

8. Pooling – это слой без обучаемых параметров, который призван сжать картинку, сохранив в ней максимум информации.

Виды pooling:

а) MaxPool – выбираем максимальное значение в пределах фильтра;

б) AvgPool – усредняем числа в пределах фильтра:

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Размер картинки после применения pooling:

$$\text{Ответ: Размер} = \frac{W-F}{S} + 1, \text{ где } F - \text{размер окна, } S - \text{шаг pooling.}$$

Padding – дополнение изображения для нормальной работы фильтра.

Виды padding:

а) Zero-padding – добавим по границам нули так, чтобы посчитанная после этого свертка давала изображение такого же размера, как и исходное. Возникает риск, что модель научится понимать, где края изображения.

б) Reflection padding – зеркальное отражение. Край картинки отзеркаливается. Не получится находить края изображения, но теперь модель может начать находить зеркальные отражения и подбирать фильтры под них.

в) Replication padding – пиксель на границе равен ближайшему пикселю из изображения. Но тогда могут получиться константные области, на которые тоже может обучиться модель.

Примеры применения техник: основные архитектуры CNN, начиная с LeNet.

9. Аугментация – преобразование входного изображения тем или иным образом, которое решает следующие задачи:

а) Снижение переобучения.

б) Условное увеличение датасета.

Способы аугментации:

а) Геометрические преобразования:

- Поворот, отражение по горизонтали/вертикали.

- Сдвиг, масштабирование (увеличение/уменьшение с обрезкой).

- Деформация (сжатие/растяжение, например, для имитации перспективы).

Плюсы: улучшают инвариантность модели к положению объектов, просты в реализации. Минусы: вертикальное отражение может искажать смысл (например, текст или логотипы).

б) Цветовые искажения:

- Коррекция яркости, контрастности, насыщенности.

- Добавление шума.

- Инверсия цветовых каналов.

Плюсы: повышают устойчивость к изменениям освещения и качества сенсоров. Минусы: риск потери информативных деталей (например, медицинские снимки).

в) Пространственные модификации:

- Random Crop.

- Padding.

Плюсы: помогают при частичной окклюзии объектов. Минусы: могут обрезать ключевые части объекта (например, лицо при детекции).

10. Отличия задач классификации, детекции и сегментации:

а) Классификация – определение класса изображения по всей фотографии.

б) Детекция – определение конкретных объектов (в т.ч. разных классов) на изображении и выделение областей, содержащих эти объекты (например, рамкой).

в) Сегментация – попиксельное выделение объектов на изображении.

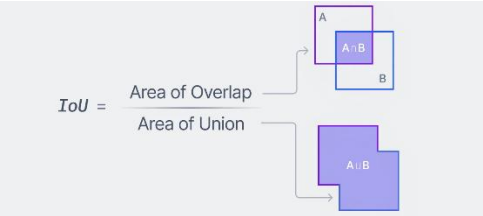
Метрика mAP:

Ответ: Усреднением AP по классам. $\text{mAP} = \frac{1}{N}\sum \text{AP}_i$, где AP — площадь под кривой precision-recall.

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

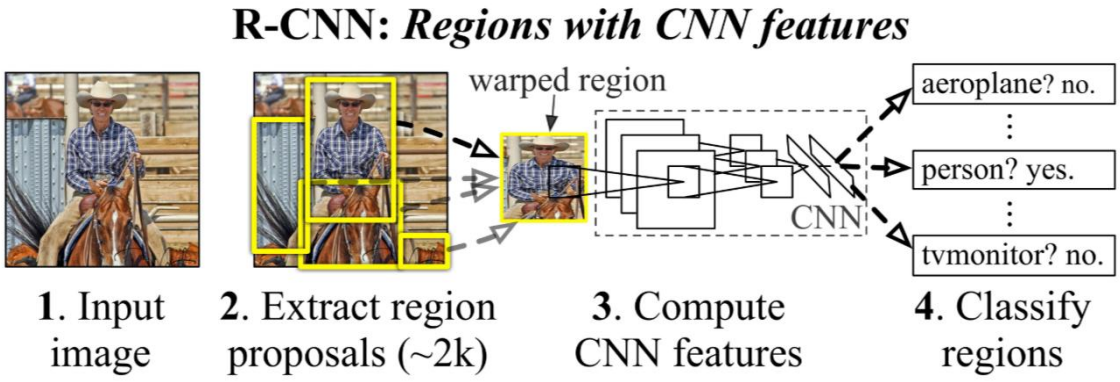
Precision и recall можно найти как IoU между истинным и выходным рамками:



Пример: IoU ≥ 0.5: TP, если класс определен верно; 0.5 > IoU: FP; IoU ≥ 0.5: FN, если класс определен неверно.

Развитие архитектур для детекции:

а) R-CNN:



- группировка областей изображения (например, по интенсивности пикселей) и их последующая склейка;

- из полученных регионов делаем bounding boxes и передаем их в предобученную сеть.

Недостатки:

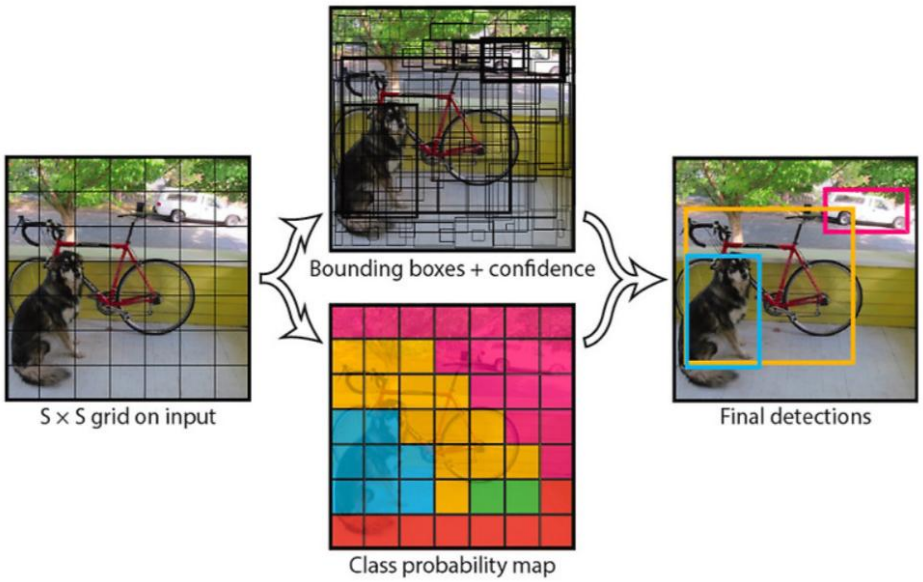
- не end-to-end: у нас несколько независимых блоков.

- генерация кандидатов bounding boxes полагается просто на интенсивность пикселей. Можно изменять алгоритмы на другие, но тогда генерация кандидатов будет очень сложной.

- сверточную сеть мы вовсе или практически не обучаем, поэтому качество вряд ли будет высокое.

- очень долго (много операций).

б) YOLO:



Решаем задачу, когда у нас C классов для детекции. Возьмем квадратное изображение и разобьем его на S·S квадратов. Будем хотеть, чтобы сетка для каждого такого квадрата нам предсказывала 5+C чисел. Первые 2 отвечают за координаты центра bbox внутри нашей клеточки. Не всей картинки, а именно нашего мини-квадрата 2 на 2. Вторые два отвечают за ширину и высоту bbox, тоже очень понятные величины. Пятое число - уверенность модели в том, что в квадратике есть какой-то объект. Если его нет, там будет что-то близкое к 0, иначе 1. Оставшиеся C чисел будут те же, что и при обычной задаче классификации: они показывают вероятности классов. Затем обучаем нейросеть (в оригинале – это архитектура DarkNet).

Достоинства:

- Обучается вся сеть целиком.

- На изображение смотрим только один раз (отсюда и название You Only Look Once). Нам не надо сначала генерировать кандидатов.

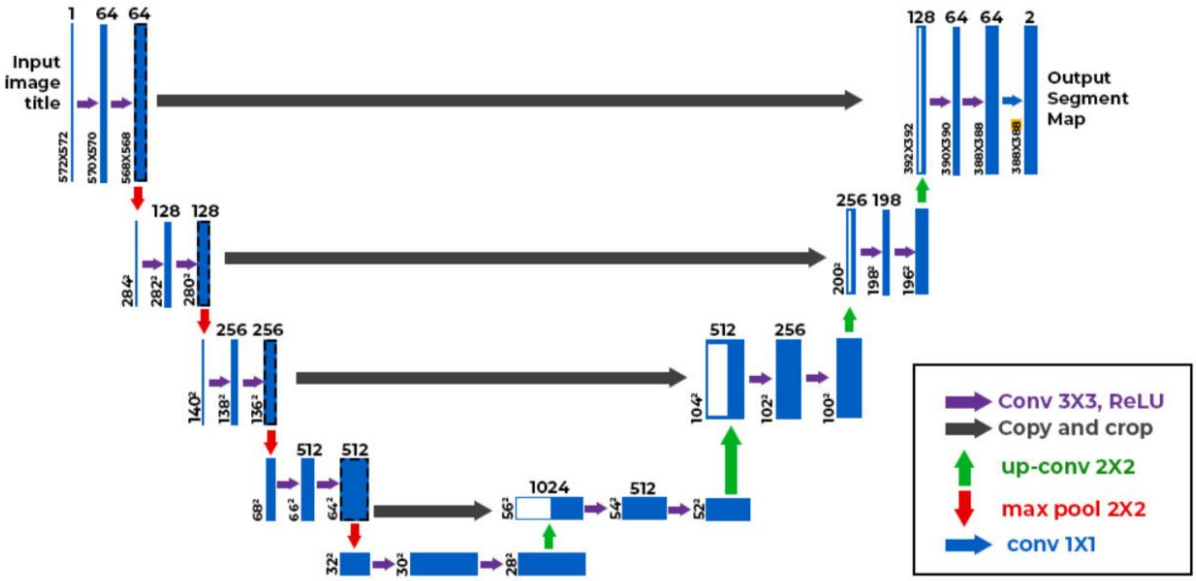
- Отсутствие разных блоков позволяет значительно ускорить модель. Современные наследники YOLO работают не только на дорогостоящих видеокартах, но и на мобильных телефонах и одноплатных компьютерах (Raspberry Pi, Orange Pi, etc.).

11. Об отличиях задач и расчет IoU см. п. 10.

Выделяют два типа сегментации: semantic, когда все объекты одного класса мы "красим одним цветом" (U-Net); и instance, когда каждый объект выделяется своим цветом и подписывается его класс (YOLO для сегментации).

Пример модели – U-Net:

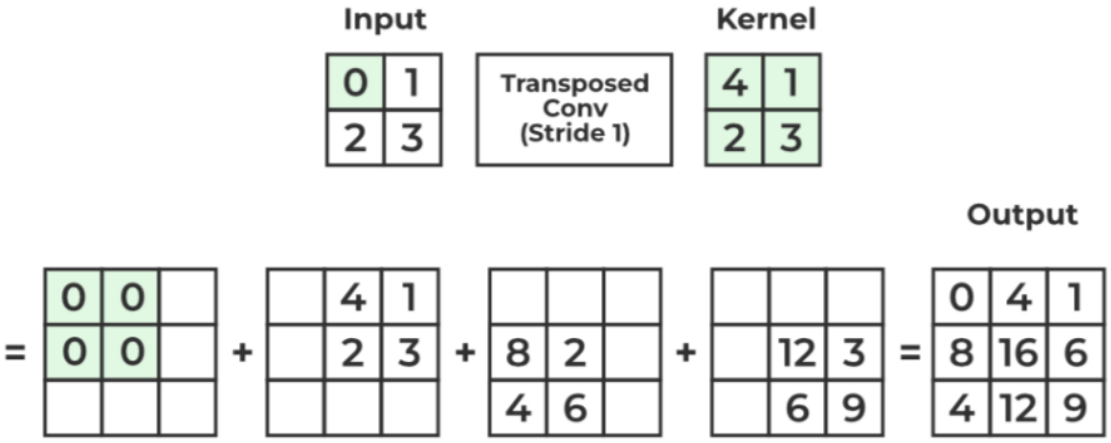
UNet



Особенности:

- copy and crop при «восстановлении» изображения;

- операция transposeConv:



Описание метрик:

Когда идет речь о метриках сегментации, самое время вспоминать метрики классификации. Задача сегментации на самом деле и есть задача классификации, просто попиксельной. То есть, все уже известные нам Accuracy, Precision и так далее будут работать. Тем не менее, чаще используют F1 меру, вспомним её формулу:

$$F_1 = \frac{2TP}{2TP + FP + FN},$$

где TP - True Positive, FP - False Positive, FN - False Negative.

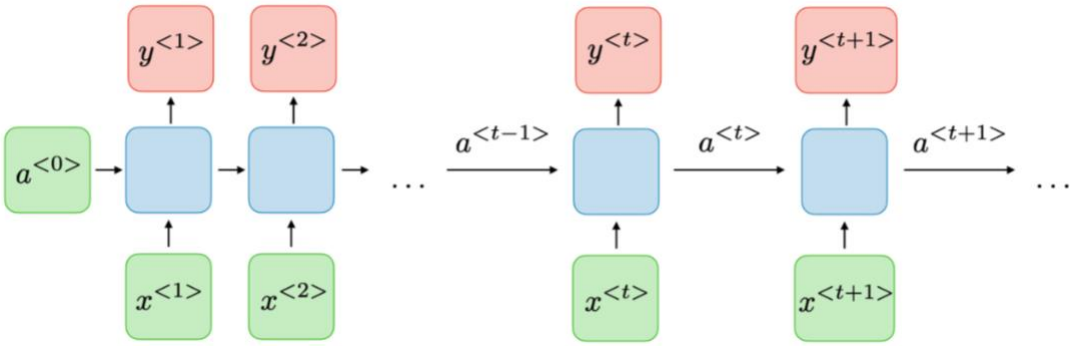
В контексте компьютерного зрения часто её называют иначе - ***Dice score***.

Другая часто используемая метрика - ***мера Жаккара***, которую можно выразить через Dice score (DSC):

$$Jac^{-1} + 1 = 2 \cdot DSC^{-1}.$$

Заметим, что если мера Жаккара близка к нулю, то Dice score будет примерно в 2 раза больше.

12. RNN читает слова (токены) текста последовательно и накапливает информацию о прочитанном в своем скрытом состоянии:



По сути, рекуррентный блок — это комбинация двух линейных (полносвзяных слоев):

а) Слой, который принимает на вход закодированный t-й токен и извлекает из него полезные признаки. Этот слой задается матрицей Wih.

б) Слой, который обновляет скрытое состояние (память) рекуррентной ячейки. Он задается матрицей весов Whh.

Формула обновления скрытого состояния:

Ответ:

$$h_t = \tanh\left(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh}\right)$$

Мотивация использования RNN заключается в том, что нам необходимо обрабатывать тексты последовательно, чтобы сохранить его структуру и смысл.

Для лучшего улавливания зависимостей в тексте иногда используются многослойные RNN.

Обучается RNN с помощью Backpropagation through time (BPTT):

Запишем формулы в следующих обозначениях:

- $a_t = g_1(x_t, a_{t-1}, w_h)$
- $\hat{y}_t = g_2(a_t, w_o)$

Мы сократили обозначения с предыдущих слайдов: веса матрицы $W_a a$ обозначили как w_h, а веса матрицы $W_a y$ обозначили как w_o.

Функция потерь вычисляется как сумма потерь по всем временным шагам:

$$L(x_1, ..., x_T, y_1, ..., y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, \hat{y}).$$

Посчитаем градиент по весу:

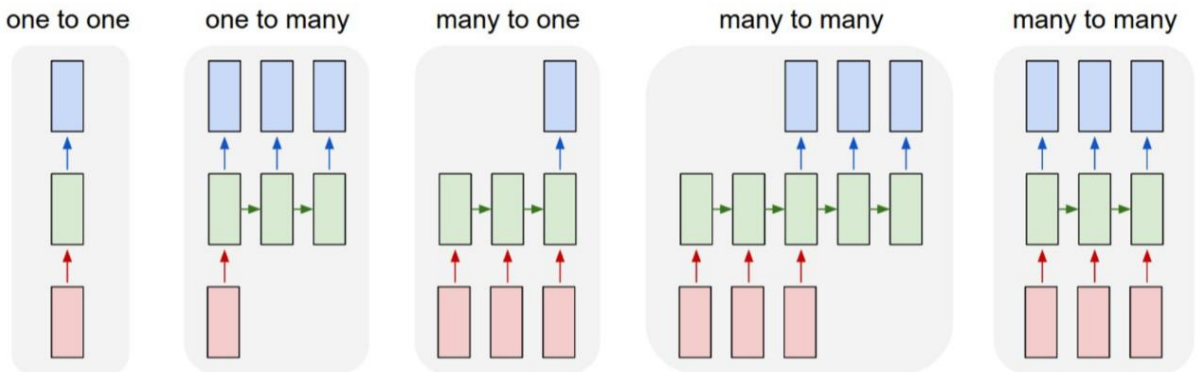
- $\frac{\partial L}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, \hat{y})}{\partial w_h} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, \hat{y})}{\partial \hat{y}} \frac{\partial g_2(a_t, w_o)}{\partial a_t} \frac{\partial a_t}{\partial w_h}$

Первый и второй множители считаются сразу, а третий придется вычислить по формуле (уходим назад во времени) - как раз здесь проявляются особенности BPTT:

- $\frac{\partial a_t}{\partial w_h} = \frac{\partial g_1(x_t, a_{t-1}, w_h)}{\partial w_h} + \frac{\partial g_1(x_t, a_{t-1}, w_h)}{\partial a_{t-1}} \frac{\partial a_{t-1}}{\partial w_h}$.

Далее, после вычисления производных, для обновления весов применяем градиентный спуск.

Виды RNN:



Проблемы при обучении RNN:

а) В силу того, что мы используем Backpropagation Through Time, то особенно остро встает проблема уменьшения и даже зануления градиентов, что делает обучение очень медленным или даже останавливает его. Это особенно проблематично при обучении долгосрочных зависимостей в последовательных данных.

б) Также при обновлении скрытого состояния сети ht мы можем потерять информацию о начале текста, особенно, длинного (информация "затрется" последними токенами текста) (решение – использование модификаций RNN вроде LSTM/GRU).

в) Возможен экспоненциальный рост и взрыв градиентов во время обратного прохода (решение – Gradient Clipping).

RNN в задачах классификации:

- анализ тональности текста (используется архитектура many-to-one);

- Part-of-speech tagger (классификация частей речи в тексте; используется архитектура many-to-many).

RNN в задачах регрессии:

- предсказание рейтинга отзыва;

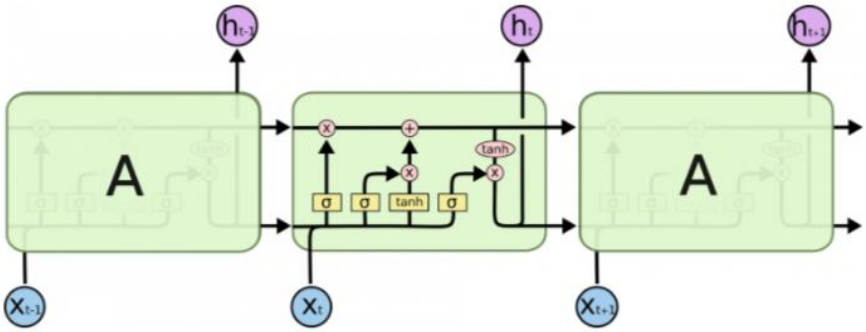
- предсказание времени чтения статьи.

13. Слабые места RNN см. в п. 12.

Классические RNN могут показывать хорошее качество только при решении задач на коротких текстовых последовательностях (не более 20-30 токенов).

Модификации:

а) LSTM (Long Short-Time Memory) – использует разделение памяти на долгосрочную и краткосрочную.



Размерность каждого гейта LSTM совпадает с размерностью скрытого состояния.

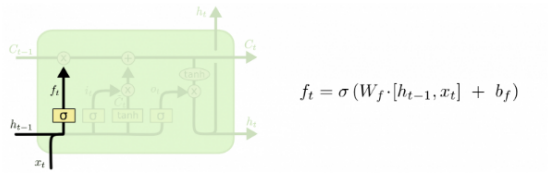
Что это такое? Долгосрочная память представлена состоянием ячейки памяти C_t. Она хранит важную информацию, которая может быть использована на более поздних временных шагах.

Как работает? Состояние ячейки памяти C_t обновляется с использованием забывающего гейта f_t и входного гейта i_t:

- Забывающий гейт контролирует, *какая часть предыдущего состояния C_{t-1} должна быть забыта*. Забывающий гейт - это вектор, который вычисляется по формуле:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f),$$

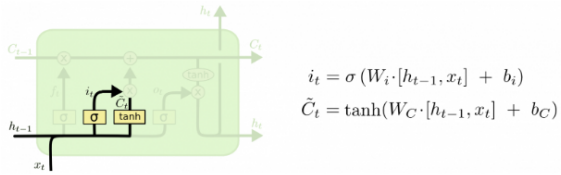
где σ - сигмоида, W_f и b_f – веса и смещение забывающего гейта.



- Входной гейт решает, *какая часть информации из поступившей на текущем шаге должна быть добавлена в долгосрочную память*. Входной гейт - это вектор, который вычисляется по формуле:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i),$$

где σ - сигмоида, W_i и b_i – веса и смещение входного гейта.



Формула обновления долгосрочной памяти:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t,$$

где

- f_t - забывающий гейт
- C_{t-1} - предыдущее состояние ячейки памяти
- i_t - входной гейт
- $\tilde{C}_t \; = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$ - вектор новой информации, часть из которой мы добавляем в долгосрочную память.

Что это такое? Краткосрочная память представлена скрытым состоянием h_t. *Это информация, которая используется для предсказаний на текущем временном шаге и передается на следующий временной шаг.*

Как работает? Краткосрочная память h_t формируется на основе обновленного состояния ячейки памяти C_t и выходного гейта o_t. Выходной гейт определяет, какая часть состояния ячейки памяти C_t будет использована для создания текущего скрытого состояния.

Формула обновления:

$$h_t = o_t * \tanh(C_t),$$

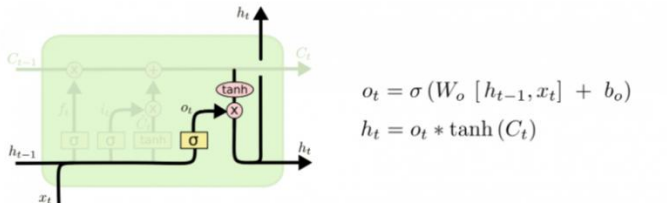
где

- o_t - выходной гейт. Выходной гейт определяет, какая часть обновленной долгосрочной памяти будет использована для создания краткосрочной памяти h_t. Выходной гейт - это вектор, который вычисляется по формуле

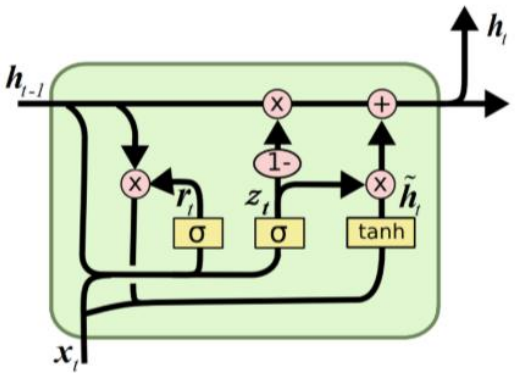
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

.

- C_t - обновленное состояние ячейки памяти.



б) GRU (Gated Recurrent Unit) предназначен для упрощения и ускорения обучения по сравнению с LSTM, сохраняя при этом большую часть ее эффективности. В отличие от LSTM, в GRU есть единый вектор состояния ячейки, без разделения на краткосрочную и долгосрочную память. Состояние ячейки в GRU является комбинацией прошлого состояния и новых входных данных, модулируемых через обновляющие и сбрасывающие ворота. Это состояние обновляется на каждом шаге и переносит информацию по всей сети.



1. **Обновляющий гейт** z_t

Определяет, какая часть предыдущего скрытого состояния h_{t-1} будет перенесена в текущее состояние h_t. Обновляющий гейт - это число, определяющееся по формуле

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z),$$

где σ — сигмоида, W_z и b_z — веса и смещение обновляющего гейта.

2. **Гейт сброса** r_t

Определяет, сколько информации из предыдущего состояния h_{t-1} будет забыто.
Формула для гейта сброса:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r),$$

где σ — сигмоида, W_r и b_r — веса и смещение гейта сброса.

3. **Кандидат на новое состояние** $\tilde{h}_t$

Определяет новое состояние, используя гейт сброса r_t для управления информацией из предыдущего состояния.
Вычисляется по формуле

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b),$$

где W и b — веса и смещение.

4. **Выходное состояние** h_t

Вычисляет обновленное состояние ячейки, комбинируя предыдущее состояние и кандидата на новое состояние с помощью обновляющего гейта:

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t,$$

где h_{t-1} — предыдущее скрытое состояние, $\tilde{h}_t$ — кандидат на новое состояние, а z_t — обновляющий гейт.

14. Эмбеддинги – это числовые вектора, кодирующие слова.

Недостатки использования простых способов векторизации (bag-of-words, tf-idf):

а) Большое число признаков в результате векторизации (а также разреженность матрицы признаков) - все это приводит к огромным временным затратам на обучение моделей, а также нередко и к переобучению

б) Похожие слова кодируются совершенно по-разному, то есть эти кодировки не сохраняют семантический смысл слов - и это для большинства задач NLP критический недостаток.

Идея word2vec: мы будем обучать такие векторы слов, чтобы слова, встречающиеся в похожих контекстах, имели близкие друг к другу векторы.

Word2Vec — это полносвязная нейронная сеть с одним скрытым слоем:

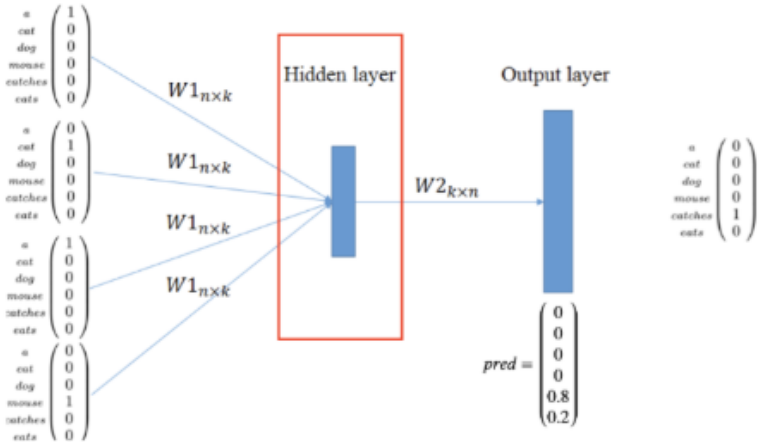
а) На вход сети подаются слова контекста, закодированные при помощи OneHot-кодирования.

б) На выходе мы получаем вектор размерности количества слов в словаре, где на i-й позиции стоит вероятность того, что внутри данного контекста стоит i-е слово из словаря.

в) На скрытом слое НЕТ функции активации, но на выходном слое функция активации - softmax - классическая функция активации в задачах многоклассовой классификации.

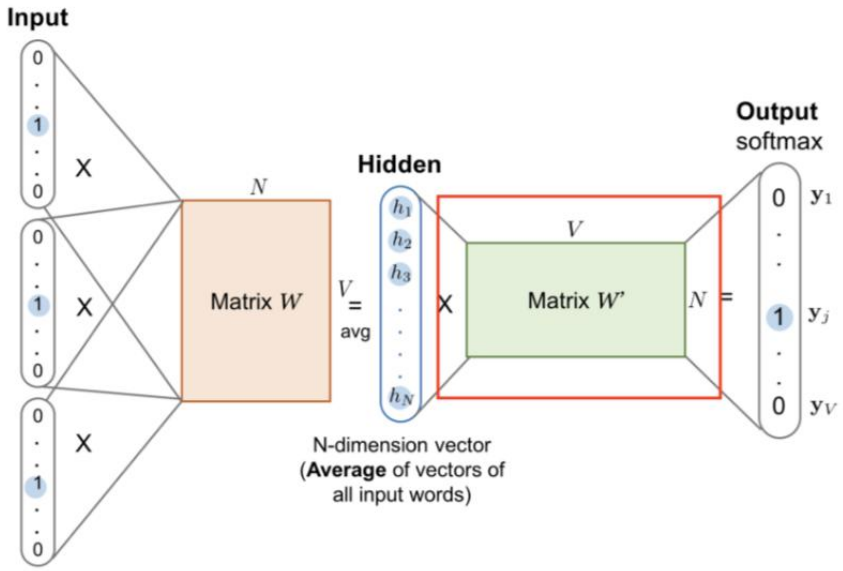
г) Функция потерь – кросс-энтропия.

д) С помощью скользящего окна движемся по тексту и набираем обучающий датасет: объекты - контекст (окружение центрального слова в окне); ответы - центральное слово.



е) поиск векторов:

Помните, что мы ищем? Мы ищем векторы слов, а где же они?



А вот где: i-**й столбец матрицы** W' **- это вектор** i**-го слова из словаря!**

Существуют и другие варианты:

- Можно взять не матрицу $W^{'}$, а матрицу W для извлечения векторов
- Можно усреднить матрицы W и $W^{'}$ и по среднему считать векторы.

Достоинства: благодаря заложенному в алгоритме предположению о том, что слова из похожих контекстов должны иметь похожий смысл, а, значит, похожие векторы - становится возможной векторная арифметика на word2vec-векторах слов.

Недостатки: не учитывает морфологию, зависит от корпуса слов; проблема многозначности слов.

В PyTorch для обучения эмбеддингов используется nn.Embedding.