Приложение № 2
к распоряжению РТУ МИРЭА
от__ _____№_____

Титульный лист материалов по дисциплине
(заполняется по каждому виду учебного материала)

ДИСЦИЛИНА Проектирование и обучение нейронных сетей
(полное наименование дисциплины без сокращений)

ИНСТИТУТ Информационые технологии

КАФЕДРА Вычислительная техника
полное наименование кафедры

ВИД УЧЕБНОГО МАТЕРИАЛА Лекция
(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ Сорокин Алексей Борисович
(фамилия, имя, отчество)

СЕМЕСТР 7 семестр, 2023/2024
(указать семестр обучения, учебный год)

ЛЕКЦИЯ. Алгоритмы с адаптивной скоростью обучения

Специалисты по нейронным сетям давно поняли, что скорость обучения – один из самых трудных для установки гиперпараметров, поскольку она существенно влияет на качество модели. Стоимость зачастую очень чувствительна в некоторых направлениях пространства параметров и нечувствительна в других. Импульсный алгоритм может в какой-то мере сгладить эти проблемы, но ценой введения другого гиперпараметра. Естественно возникает вопрос, нет ли какого-то иного способа. Если мы полагаем, что направления чувствительности почти параллельны осям, то, возможно, имеет смысл задавать скорость обучения отдельно для каждого параметра и автоматически адаптировать эти скорости на протяжении всего обучения.

Алгоритм delta-bar-delta – один из первых эвристических подходов к адаптации индивидуальных скоростей обучения параметров модели. Он основан на простой идее: если частная производная функции потерь по данному параметру модели не меняет знак, то скорость обучения следует увеличить. Если же знак меняется, то скорость следует уменьшить. Конечно, такого рода правило применимо только к оптимизации на полном пакете. Позже был предложен целый ряд инкрементных (увеличивающих и основанных на мини-пакетах) методов для адаптации скоростей обучения параметров.

AdaGrad

Алгоритм AdaGrad по отдельности адаптирует скорости обучения всех параметров модели, умножая их на коэффициент, обратно пропорциональный квадратному корню из суммы всех прошлых значений квадрата градиента. Для параметров, по которым частная производная функции потерь наибольшая, скорость обучения уменьшается быстро, а если частная производная мала, то и скорость обучения уменьшается медленнее. В итоге больший прогресс получается в направлениях пространства параметров со сравнительно пологими склонами. В случае выпуклой оптимизации у алгоритма AdaGrad есть некоторые желательные теоретические свойства. Но эмпирически при обучении глубоких нейронных сетей накапливание квадратов градиента с самого начала обучения может привести к преждевременному и чрезмерному уменьшению эффективной скорости обучения. AdaGrad хорошо работает для некоторых, но не для всех моделей глубокого обучения.

Алгоритм 3. Алгоритм AdaGrad

Require: глобальная скорость обучения ε

Require: начальные значения параметров θ

Require: небольшая константа δ, например 10^{-7}, для обеспечения

численной устойчивости

Инициализировать переменную для агрегирования градиента $r = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)};\, \theta), y^{(i)})$.

Агрегировать квадраты градиента: $r \leftarrow r + g \odot g$.

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta+\sqrt{r}} \odot g$ (операции деления и извлечения корня применяются к каждому элементу).

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

$\odot$ - N-арный оператор. Операция n-арная на множестве g — отображение $f: g^n \to g$, где g^n — декартова n-я степень множества g.

RMSProp

Алгоритм RMSProp – это модификация AdaGrad, призванная улучшить его поведение в невыпуклом случае путем изменения способа агрегирования градиента на экспоненциально взвешенное скользящее среднее. AdaGrad разрабатывался для быстрой сходимости в применении к выпуклой функции. Если же он применяется к невыпуклой функции для обучения нейронной сети, то траектория обучения может проходить через много разных структур и в конечном итоге прийти в локально выпуклую впадину. AdaGrad уменьшает скорость обучения, принимая во внимание всю историю квадрата градиента, и может случиться так, что скорость станет слишком малой еще до достижения такой выпуклой структуры. В алгоритме RMSProp используется экспоненциально затухающее среднее, т. е. далекое прошлое отбрасывается, чтобы повысить скорость сходимости после обнаружения выпуклой впадины, как если бы внутри этой впадины алгоритм AdaGrad был инициализирован заново.

Алгоритм 4. содержит описание RMSProp в стандартной форме. По сравнению с AdaGrad вводится новый гиперпараметр ρ, управляющий масштабом длины при вычислении скользящего среднего.

Алгоритм 4. Алгоритм RMSProp

Require: глобальная скорость обучения ε, скорость затухания ρ

Require: начальные значения параметров θ

Require: небольшая константа δ, например 10^{-6}, для стабилизации деления на малые числа

Инициализировать переменную для агрегирования градиента $r = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)};\, \theta), y^{(i)})$.

Агрегировать квадраты градиента: $r \leftarrow \rho r + (1-\rho) g \odot g$.

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta+\sqrt{r}} \odot g$ (операции деления и извлечения корня применяются к каждому элементу).

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Эмпирически показано, что RMSProp – эффективный и практичный алгоритм оптимизации глубоких нейронных сетей. В настоящее время он считается одним из лучших методов оптимизации и постоянно используется в практической работе.

Adam

Adam – еще один алгоритм оптимизации с адаптивной скоростью обучения. Название «Adam» – сокращение от «adaptive moments» (адаптивные моменты). Его, наверное, правильнее всего рассматривать как комбинацию RMSProp и импульсного метода с несколькими важными отличиями. Во-первых, в Adam импульс включен непосредственно в виде оценки первого момента (с экспоненциальными весами) градиента. Самый прямой способ добавить импульс в RMSProp – применить его к масштабированным градиентам. У использования импульса в сочетании с масштабированием нет ясного теоретического обоснования. Во-вторых, Adam включает поправку на смещение в оценки как первых моментов (член импульса), так и вторых (нецентрированных) моментов для учета их инициализации в начале координат (см. алгоритм 8.7). RMSProp также включает оценку (нецентрированного) второго момента, однако в нем нет поправочного коэффициента. Таким образом, в отличие от Adam, в RMSProp оценка второго момента может иметь высокое смещение на ранних стадиях обучения. Вообще говоря, Adam считается довольно устойчивым к выбору гиперпараметров, хотя скорость обучения иногда нужно брать отличной от предлагаемой по умолчанию.

Алгоритм 5. Алгоритм Adam

Require: величина шага ε (по умолчанию 0.001).

Require: коэффициенты экспоненциального затухания для оценок моментов ρ_1 и ρ_2, принадлежащие диапазону $[0, 1)$ (по умолчанию 0.9 и 0.999

соответственно).

Require: небольшая константа δ для обеспечения численной устойчивости (по умолчанию 10^{-8}).

Require: начальные значения параметров θ.

Инициализировать переменные для первого и второго моментов $s = 0$, $r = 0$

Инициализировать шаг по времени $t = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)}; \theta), y^{(i)})$.

$t \leftarrow t + 1$

Обновить смещенную оценку первого момента: $s \leftarrow \rho_1 s + (1- \rho_1)g$

Обновить смещенную оценку второго момента: $r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$

Скорректировать смещение первого момента: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Скорректировать смещение второго момента: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\hat{s}}{\delta+\sqrt{\hat{r}}}$ (операции применяются к каждому элементу)

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Выбор правильного алгоритма оптимизации

Таким образом, обсудили ряд родственных алгоритмов, каждый из которых пытается решить проблему оптимизации глубоких моделей, адаптируя скорость обучения каждого параметра. Возникает естественный вопрос: какой алгоритм выбрать? К сожалению, в настоящее время единого мнения нет. Результаты показывают, что семейство алгоритмов с адаптивной скоростью обучения (представленное алгоритмами RMSProp и AdaDelta) ведет себя достаточно устойчиво, явный победитель здесь не выявлен. Сейчас наиболее популярны и активно применяются алгоритмы СГС, СГС с учетом импульса, RMSProp, RMSProp с учетом импульса, AdaDelta и Adam. Какой из них использовать, зависит главным образом от знакомства пользователя с алгоритмом.

Приближенные методы второго порядка

Теперь обсудим применение методов второго порядка к обучению глубоких сетей. Для простоты будем рассматривать только одну целевую

функцию: эмпирический риск:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data(x,y)}}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})$$

Впрочем, рассматриваемые методы легко обобщаются на другие целевые функции.

Метод Ньютона

Мы с Вами познакомились с градиентными методами второго порядка. И знаем, что в отличие от методов первого порядка, в этом случае для улучшения оптимизации задействуются вторые производные. Самый известный метод второго порядка – метод Ньютона. Опишем его более подробно с акцентом на применении к обучению нейронных сетей.

Метод Ньютона основан на использовании разложения в ряд Тейлора с точностью до членов второго порядка для аппроксимации $J(\theta)$ в окрестности некоторой точки θ_0, производные более высокого порядка при этом игнорируются.

$$J(\theta) \approx J(\theta_0 0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + 1/2(\theta - \theta_0)^T H(\theta - \theta_0)$$

где H – гессиан J относительно θ, вычисленный в точке θ_0. Пытаясь найти критическую точку этой функции, приходим к правилу Ньютона для обновления параметров:

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

Таким образом, для локально квадратичной функции (с положительно определенной матрицей H) умножение градиента на H^{-1} сразу дает точку минимума. Если целевая функция выпуклая, но не квадратичная (имеются члены более высокого порядка), то это обновление можно повторить, получив тем самым алгоритм обучения, основанный на методе Ньютона.

Для неквадратичных поверхностей метод Ньютона можно применять итеративно, при условии что матрица Гессе остается положительно определенной. Отсюда вытекает двухшаговая итеративная процедура. Сначала обновляем или вычисляем обратный гессиан (путем обновления квадратичной аппроксимации). Затем обновляем параметры в соответствии с правилом Ньютона.

Алгоритм 6. Метод Ньютона с целевой функций

$$\boldsymbol{J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})}$$

Require: начальные значения параметров θ_0.

Require: обучающий набор m примеров

while критерий остановки не выполнен **do**

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

Вычислить гессиан: $H \leftarrow (1/m)\nabla_\theta^2 \sum_i L(f(x^{(i)};\theta), y^{(i)})$

Вычислить обратный гессиан: H^{-1}

Вычислить обновление: $\Delta\theta = -H^{-1}g$

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Вспомним, что метод Ньютона применим, только если матрица Гессе положительно определенная. В глубоком обучении поверхность целевой функции обычно невыпуклая и имеет много особенностей типа седловых точек, с которыми метод Ньютона не справляется. Если не все собственные значения гессиана положительны, например вблизи седловой точки, то метод Ньютона может произвести обновление не в том направлении. Такую ситуацию можно предотвратить с помощью регуляризации гессиана. Одна из распространенных стратегий регуляризации – прибавление константы α ко всем диагональным элементам гессиана. Тогда регуляризированное обновление принимает вид:

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1}\nabla_\theta f(\theta_0)$$

Эта стратегия регуляризации применяется в аппроксимациях метода Ньютона, например в алгоритме Левенберга–Марквардта, и работает неплохо, если отрицательные собственные значения гессиана сравнительно близки к нулю. Если же в некоторых направлениях кривизна сильнее, то значение α следует выбирать достаточно большим для компенсации отрицательных собственных значений. Однако по мере увеличения α в гессиане начинает доминировать диагональ αI, и направление, выбранное методом Ньютона, стремится к стандартному градиенту, поделенному на α. При наличии сильной кривизны α должно быть настолько большим, чтобы метод Ньютона делал меньшие шаги, чем градиентный спуск с подходящей скоростью обучения. Помимо проблем, связанных с такими особенностями целевой функции, как седловые точки, применение метода Ньютона к обучению больших нейронных сетей лимитируется требованиями к вычислительным ресурсам. Число элементов гессиана равно квадрату числа параметров, поэтому при k параметрах метод Ньютона требует обращения матрицы размера $k \times k$. Кроме того, поскольку параметры изменяются при каждом обновлении, обратный гессиан нужно вычислять на каждой итерации обучения. Поэтому лишь сети с очень небольшим числом параметров реально обучить методом Ньютона.

Метод сопряженных градиентов

Метод сопряженных градиентов позволяет избежать вычисления обратного гессиана посредством итеративного спуска в сопряженных направлениях. Идея этого подхода вытекает из внимательного изучения слабого места метода наискорейшего спуска, при котором поиск итеративно производится в направлении градиента. На рис. 3 показано, что метод наискорейшего спуска в квадратичной впадине неэффективен, т. к. продвигается зигзагами. Так происходит, потому что направление линейного поиска, определяемое градиентом на очередном шаге, гарантированно ортогонально направлению поиска на предыдущем шаге.

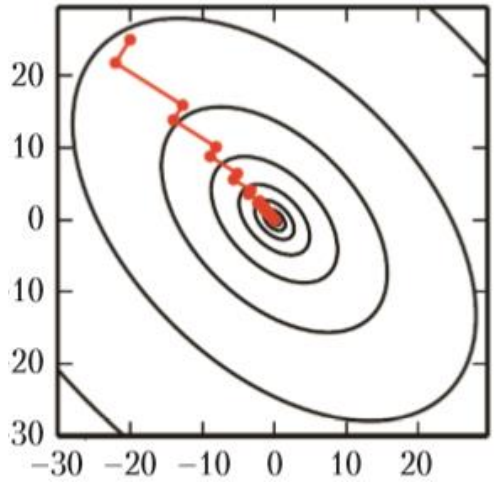

Рис. 3. Метод наискорейшего спуска в применении к поверхности квадратичной целевой функции.

В этом методе на каждом шаге производится переход в точку с наименьшей стоимостью вдоль прямой, определяемой градиентом в начале этого шага. Это решает некоторые показанные на рис. 2 проблемы, которые обусловлены фиксированной скоростью обучения, но даже при оптимальной величине шага алгоритм все равно продвигается к оптимуму зигзагами. По определению, в точке минимума целевой функции вдоль заданного направления градиент в конечной точке ортогонален этому направлению.

Обозначим d_{t-1} направление предыдущего поиска. В точке минимума, где поиск завершается, производная по направлению d_{t-1} равна нулю: $\nabla\theta J(\theta) \times d_{t-1} = 0$. Поскольку градиент в этой точке определяет текущее направление поиска, $dt = \nabla_\theta J(\theta)$ не дает вклада в направлении d_{t-1}. Следовательно, dt ортогонально d_{t-1}. Эта связь между d_{t-1} и dt иллюстрируется на рис. 3 для нескольких итераций метода наискорейшего спуска. Как видно по рисунку, выбор ортогональных направлений спуска не сохраняет минимума вдоль предыдущих направлений поиска. Отсюда и зигзагообразная траектория, поскольку после спуска к минимуму в направлении текущего градиента должны

заново минимизировать целевую функцию в направлении предыдущего градиента. А значит, следуя направлению градиента в конце каждого отрезка, в некотором смысле перечеркивается достигнутое в направлении предыдущего отрезка. Метод сопряженных градиентов и призван решить эту проблему.

В методе сопряженных градиентов направление следующего поиска является сопряженным к направлению предыдущего, т. е. не отказывается от того, что было достигнуто в предыдущем направлении. На t-ой итерации обучения направление следующего поиска определяется формулой:

$$dt = \nabla_\theta J(\theta) + \beta_t d_{t-1},$$

где β_t – коэффициент, определяющий, какую часть направления d_{t-1} следует прибавить к текущему направлению поиска.

Два направления dt и d_{t-1} называются сопряженными, если $d_t^T H d_{t-1} = 0$, где H – матрица Гессе. Самый простой способ обеспечить сопряженность – вычислить собственные векторы H для выбора β_t – не отвечает исходной цели разработать метод, который был бы вычислительно проще метода Ньютона при решении больших задач. Можно ли найти сопряженные направления, не прибегая к таким вычислениям? К счастью, да. Существуют два популярных метода вычисления β_t.

1. Метод Флетчера-Ривса:

$$\beta_t = \frac{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

2. Метод Полака-Рибьера:

$$\beta_t = \frac{(\nabla_\theta J(\theta_t) - \nabla_\theta J(\theta_{t-1}))^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

Для квадратичной поверхности сопряженность направлений гарантирует, что модуль градиента вдоль предыдущего направления не увеличится. Поэтому минимум, найденный вдоль предыдущих направлений, сохраняется. Следовательно, в k-мерном пространстве параметров метод сопряженных градиентов требует не более k поисков для нахождения минимума.

Нелинейный метод сопряженных градиентов. Мы обсуждали метод сопряженных градиентов в применении к квадратичной целевой функции. Но в нас интересуют в основном методы оптимизации для обучения нейронных сетей и других глубоких моделей, в которых целевая функция далека от квадратичной. Как ни странно, метод сопряженных градиентов применим и в такой ситуации, хотя и с некоторыми изменениями. Если целевая функция не квадратичная, то уже нельзя гарантировать, что поиск в сопряженном направлении сохраняет минимум в предыдущих направлениях. Поэтому в нелинейном алгоритме

сопряженных градиентов время от времени производится сброс, когда метод сопряженных градиентов заново начинает поиск вдоль направления неизмененного градиента.

Алгоритм 7. Метод сопряженных градиентов

Require: начальные значения параметров θ_0.

Require: обучающий набор m примеров

Инициализировать $\rho_0 = 0$

Инициализировать $g_0 = 0$

Инициализировать $t = 1$

while критерий остановки не выполнен **do**

Инициализировать градиент $g_t = 0$

Вычислить градиент: $g_t \leftarrow (1/m)\nabla_\theta \sum_i L(f(x^{(i)};\ \theta), y^{(i)})$

Вычислить $\beta_t = ((g_t - g_{t-1})^T g_t)/g_{t-1}^T g_{t-1}$ (метод Полака–Рибьера)

Вычислить направление поиска: $\rho_t = -g_t + \beta_t \rho_{t-1}$

Произвести поиск с целью нахождения: $\varepsilon^* = argmin_\varepsilon (1/m) \sum_{i=1}^m L(f(x^{(i)};\ \theta_t + \varepsilon\rho_t), y^{(i)})$

Применить обновление: $\theta_{t+1} = \theta_t + \varepsilon^* \rho_t$

$t \leftarrow t + 1$

end while

Есть сообщения, что нелинейный метод сопряженных градиентов дает неплохие результаты при обучении нейронных сетей, хотя часто имеет смысл инициализировать оптимизацию, выполнив несколько итераций стохастического градиентного спуска, и только потом переходить к нелинейным сопряженным градиентам. Кроме того, хотя нелинейный алгоритм сопряженных градиентов традиционно считался пакетным методом, его мини-пакетные варианты успешно применялись для обучения нейронных сетей. Позже были предложены адаптации метода сопряженных градиентов, например алгоритм масштабированных сопряженных градиентов.

Алгоритм BFGS

Алгоритм Бройдена–Флетчера–Гольдфарба–Шанно (BFGS) – попытка взять некоторые преимущества метода Ньютона без обременительных вычислений. В этом смысле BFGS аналогичен методу сопряженных градиентов. Однако в BFGS подход к аппроксимации обновления Ньютона более прямолинейный. Напомним, что обновление Ньютона определяется формулой

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

где H – гессиан J относительно θ, вычисленный в точке θ_0. Основная вычислительная трудность при применении обновления Ньютона – вычисление

обратного гессиана H^{-1}. В квазиньютоновских методах (из которых алгоритм BFGS самый известный) обратный гессиан аппроксимируется матрицей M_t, которая итеративно уточняется в ходе обновлений низкого ранга.

После нахождения аппроксимации гессиана M_t направление спуска ρ_t определяется по формуле $\rho_t = M_t g_t$. В этом направлении производится линейный поиск для определения величины шага ε^*. Окончательное обновление параметров производится по формуле

$$\theta_{t+1} = \theta_t + \varepsilon^* \rho_t$$

Как и в методе сопряженных градиентов, в алгоритме BFGS производится последовательность линейных поисков в направлениях, вычисляемых с учетом информации второго порядка. Но, в отличие от метода сопряженных градиентов, успех не так сильно зависит от того, находит ли линейный поиск точку, очень близкую к истинному минимуму вдоль данного направления. Поэтому BFGS тратит меньше времени на уточнение результатов каждого линейного поиска. С другой стороны, BFGS должен хранить обратный гессиан M, для чего требуется память объема $O(n^2)$, поэтому BFGS непригоден для современных моделей глубокого обучения, насчитывающих миллионы параметров.

BFGS в ограниченной памяти (L-BFGS). Потребление памяти в алгоритме BFGS можно значительно уменьшить, если не хранить полную аппроксимацию обратного гессиана M. В алгоритме L-BFGS аппроксимация M вычисляется так же, как в BFGS, но, вместо того чтобы сохранять аппроксимацию между итерациями, делается предположение, что $M^{(t-1)}$ – единичная матрица. При использовании совместно с точным линейным поиском направления, вычисляемые алгоритмом L-BFGS, являются взаимно сопряженными. Однако, в отличие от метода сопряженных градиентов, эта процедура ведет себя хорошо, даже когда линейный поиск находит только приближенный минимум. Описанную стратегию L-BFGS без запоминания можно обобщить, включив больше информации о гессиане; для этого нужно хранить некоторые векторы, используемые для обновления M на каждом шаге, тогда потребуется только память объемом $O(n)$.

Стратегии оптимизации и метаалгоритмы

Многие методы оптимизации – не совсем алгоритмы, а скорее общие шаблоны, которые можно специализировать и получить алгоритмы или подпрограммы, включаемые в различные алгоритмы.

Пакетная нормировка

Пакетная нормировка – одна из наиболее интересных новаций в области оптимизации глубоких нейронных сетей – вообще алгоритмом не является. Это

метод адаптивной перепараметризации (функция называется заменой параметра), появившийся из-за трудностей обучения очень глубоких моделей.

Пакетная нормализация (англ. batch-normalization) — метод, который позволяет повысить производительность и стабилизировать работу искусственных нейронных сетей. Суть данного метода заключается в том, что некоторым слоям нейронной сети на вход подаются данные, предварительно обработанные и имеющие нулевое математическое ожидание и единичную дисперсию.

Нормализация входного слоя нейронной сети обычно выполняется путем масштабирования данных, подаваемых в функции активации. Например, когда есть признаки со значениями от 0 до 1 и некоторые признаки со значениями от 1 до 1000, то их необходимо нормализовать, чтобы ускорить обучение. Нормализацию данных можно выполнить и в скрытых слоях нейронных сетей, что и делает метод пакетной нормализации.

Пакетная нормализация уменьшает величину, на которую смещаются значения узлов в скрытых слоях (т.н. ковариантный сдвиг). Ковариантный сдвиг — это ситуация, когда распределения значений признаков в обучающей и тестовой выборке имеют разные параметры (математическое ожидание, дисперсия и т.д.). Ковариантность в данном случае относится к значениям признаков.

Проиллюстрируем ковариантный сдвиг примером. Пусть есть глубокая нейронная сеть, которая обучена определять находится ли на изображении роза. И нейронная сеть была обучена на изображениях только красных роз. Теперь, если попытаться использовать обученную модель для обнаружения роз различных цветов, то, очевидно, точность работы модели будет неудовлетворительной. Это происходит из-за того, что обучающая и тестовая выборки содержат изображения красных роз и роз различных цветов в разных пропорциях. Другими словами, если модель обучена отображению из множества X в множество Y и если пропорция элементов в X изменяется, то появляется необходимость обучить модель заново, чтобы «выровнять» пропорции элементов в X и Y. Когда пакеты содержат изображения разных классов, распределенные в одинаковой пропорции на всем множестве, то ковариантный сдвиг незначителен. Однако, когда пакеты выбираются только из одного или двух подмножеств (в данном случае, красные розы и розы различных цветов), то ковариантный сдвиг возрастает. Это довольно сильно замедляет процесс обучения модели.

Простой способ решить проблему ковариантного сдвига для входного слоя

— это случайным образом перемешать данные перед созданием пакетов. Но для скрытых слоев нейронной сети такой метод не подходит, так как распределение входных данных для каждого узла скрытых слоев изменяется каждый раз, когда происходит обновление параметров в предыдущем слое. Эта проблема называется внутренним ковариантным сдвигом. Для решения данной проблемы часто приходится использовать низкий темп обучения и методы регуляризации при обучении модели. Другим способом устранения внутреннего ковариантного сдвига является метод пакетной нормализации.

Описание метода

Опишем устройство метода пакетной нормализации. Пусть на вход некоторому слою нейронной сети поступает вектор размерности d: $x = (x^{(1)}, \dots, x^{(d)})$. Нормализуем данный вектор по каждой размерности k:

$$\hat{x}^{(1)} = \frac{x^{(k)} - E\left(x^{(k)}\right)}{\sqrt{D(x^{(k)})}}$$

где математическое ожидание и дисперсия считаются по всей обучающей выборке. Такая нормализация входа слоя нейронной сети может изменить представление данных в слое. Чтобы избежать данной проблемы, вводятся два параметра сжатия и сдвига нормализованной величины для каждого $x^{(k)}$: $\gamma^{(k)}$, $\beta^{(k)}$ — которые действуют следующим образом:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Данные параметры настраиваются в процессе обучения вместе с остальными параметрами модели. Пусть обучение модели производится с помощью пакетов B размера m: $B = \{x_1, \dots, x_m\}$. Здесь нормализация применяется к каждому элементу входа с номером k отдельно, поэтому в $x^{(k)}$ индекс опускается для ясности изложения. Пусть были получены нормализованные значения пакета $\hat{x}_1, \dots, \hat{x}_m$. После применения операций сжатия и сдвига были получены $y_1, \dots, y_m$. Обозначим данную функцию пакетной нормализации следующим образом:

$$BN_{\gamma,\beta}: \{x_1, \dots, x_m\} \to \{y1, \dots, ym\}$$

Тогда алгоритм пакетной нормализации можно представить так:

Алгоритм 8. Пакетной нормализации

Вход: значения x из пакета $B = \{x_1, \dots, x_m\}$; настраиваемые параметры γ, β; константа ϵ для вычислительной устойчивости.

Выход: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_B = \frac{1}{m}\sum_i^m x_i$ ***// математическое ожидание пакета***

$\sigma_B^2 = \frac{1}{m}\sum_i^m (x_i - \mu_B)^2$ // ***дисперсия пакета***

$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ // ***нормализация***

$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ // ***сжатие и сдвиг***

Заметим, что если $\beta = \mu_B$ и $\gamma = \sqrt{\sigma_B^2 + \epsilon}$, то y_i равен x_i, то есть $BN_{\gamma,\beta}(\cdot)$ является тождественным отображением. Таким образом, использование пакетной нормализации не может привести к снижению точности, поскольку оптимизатор просто может использовать нормализацию как тождественное отображение.

Ипользование пакетной нормализации обладает еще несколькими дополнительными полезными свойствами:

- достигается более быстрая сходимость моделей, несмотря на выполнение дополнительных вычислений;
- пакетная нормализация позволяет каждому слою сети обучаться более независимо от других слоев;
- становится возможным использование более высокого темпа обучения, так как пакетная нормализация гарантирует, что выходы узлов нейронной сети не будут иметь слишком больших или малых значений;
- пакетная нормализация в каком-то смысле также является механизмом регуляризации: данный метод привносит в выходы узлов скрытых слоев некоторый шум, аналогично методу dropout (дропаут (англ. dropout) — метод регуляризации нейронной сети для предотвращения переобучения);
- модели становятся менее чувствительны к начальной инициализации весов.

КАПСУЛЬНАЯ НЕЙРОННАЯ СЕТЬ

В 2017 году Джеффри Хинтон (один из основоположников подхода обратного распространения ошибки) опубликовал статью, в которой описал капсульные нейронные сети и предложил алгоритм динамической маршрутизации между капсулами для обучения предложенной архитектуры. По словам автора, особенности данной архитектуры направлены на решение проблем одного из самых применяющихся к данной задачи методов – сверточных нейронных сетей. Как Вы уже знаете, данная сеть обычно состоит из множества чередующихся слоев трех основных видов: сверточного, субдискретизирующего (пулинга) и полносвязного

Хинтон указывает на три основных недостатка архитектуры сверточных нейронных сетей:

- Сверточные сети в своей структуре имеют слишком малое количество уровней: из нейронов складываются слои, из слоев – сеть. Необходимо, некоторым образом, объединить нейроны в группы на каждом слое так, чтобы появилась возможность производить внутренние вычисления и выдавать на выходе компактный результат;
- По мере продвижения от первых слоев сверток, где выделяются локальные признаки исходного изображения, к более глубоким слоям признаки превращаются в так называемые домены признаков, что является одним из главных достоинств сверточных нейронных сетей. При этом критике подвергается только слой пулинга: на каждом таком слое забывается информация о локации выделенного признака, что плохо вписывается в механизм восприятия формы человеческого мозга. Исчезают пространственные связи между объектами или их частями;
- Наличие слоев пулинга также приводит к небольшой трасляционной инвариантности, которая по мнению Хинтона является недостатком сети, а не достоинством. Проблема заключается в неспособности сети определять положение объекта в пространстве, а также реагировать на его изменения (такие как поворот или смещение). Для решения данного вопроса часто применяется аугментация данных – расширение обучающей выборки, заключающееся в создании дополнительных данных из уже имеющихся путем различных преобразований: поворота, отражения, масштабирования, изменения цвета. Это приводит к увеличению входного набора, а значит и длительности процесса обучения. Вместо инвариантности автор советует прийти к эквивариантности, т.е. пониманию свойств объекта таким образом, что при их изменении меняется соответственно и результат представления сети о данном объекте.

Инвариант является свойством объекта, которое не изменяется в результате некоторого преобразования. Например, площадь круга не изменится, если круг сдвинуть влево.

Неформально эквивариантность - это свойство, которое предсказуемо изменяется при преобразовании. Например, при смещении центр круга перемещается на ту же величину, что и круг. Неэквивариантность - это свойство, значение которого не изменяется предсказуемо при преобразовании. Например, преобразование круга в эллипс означает, что его периметр больше не может быть вычислен как π, умноженное на диаметр. В компьютерном зрении ожидается, что класс объекта будет инвариантом для многих преобразований. То есть кошка остается кошкой, если ее сдвинуть, перевернуть вверх ногами или уменьшить в размерах. Однако многие другие свойства вместо этого эквивалентны. Объем

кота меняется при масштабировании.

Эквивариантные свойства, такие как пространственная взаимосвязь фиксируются в позе, данные, описывающий объект перевод, вращение , масштаб и отражение. Перевод - это изменение местоположения в одном или нескольких измерениях. Вращение - это изменение ориентации. Масштаб - это изменение размера. Отражение - это зеркальное отображение. Неконтролируемые сети изучают глобальное линейное многообразие между объектом и его позицией в виде матрицы весов. Другими словами, капсульные нейронные сети могут идентифицировать объект независимо от его позы, вместо того, чтобы учиться распознавать объект, включая его пространственные отношения как часть объекта. В сетке поза может включать свойства, отличные от пространственных отношений, например, цвет (кошки могут быть разных цветов).

Умножение объекта на многообразие создает объект (для объекта в пространстве).

Предположим, что на изображении присутствуют два одинаковых признака. В результате операции свертки, на выходе в соответствующих местах следует ожидать схожие значения. Однако, при использовании слоя глобальной агрегации средних значений (англ. global average pooling), в пределах окна фиксированного размера или полносвязных слоев в конце архитектуры, теряется эквивариантность в пользу инвариантности признаков. Ко всему прочему, после прохождения через слои субдискретизации, теряется доля информации о нахождении этих признаков на изображении. Причина данной проблемы заключается в механизме слоев субдискретизации (рис. 1).

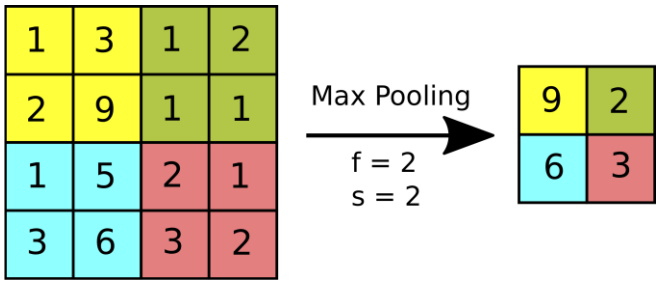

Рис. 1. Принцип работы слоя субдискретизации максимальных значений.

Если у объектов на двух изображениях признаки будут отличаться поворотом или размером, то, в результате, два объекта могут быть не распознаны как объекты одного класса (рис. 2), если не применить аугментацию данных.

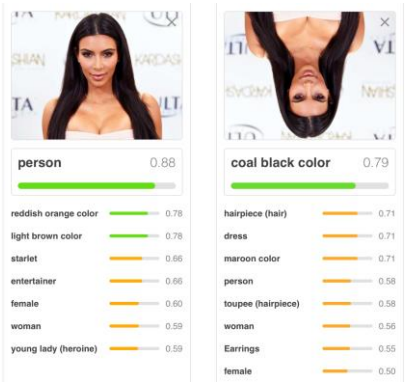

Рис. 2. Визуализация проблемы с классификацией двух изображений

И, напротив, сверточная сеть может воспринимать изображение, содержащее разбросанные признаки объекта, как сам объект. Визуализация данной проблемы представлена на рис. 3.

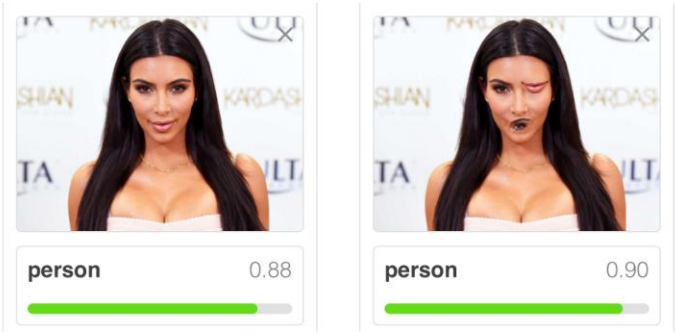

Рис. 3. Визуализация проблемы с классификацией набора признаков как объект

Таким образом, capsule Neural Network (CapsNet) представляет собой систему машинного обучения, которая является одним из видов искусственной нейронной сети (ИНС), которые могут быть использованы для более модели иерархических отношений. Подход представляет собой попытку более точно имитировать биологическую нейронную организацию. Идея состоит в том, чтобы добавить структуры, называемые «капсулами», в сверточную нейронную сеть (CNN) и повторно использовать выходные данные нескольких из этих капсул для формирования более стабильных (по отношению к различным возмущениям) представлений для более высоких капсул. Результатом является вектор, состоящий из вероятности наблюдения и позы для этого наблюдения. Этот вектор аналогичен тому, что делается, например, при классификации с локализацией в CNN.

Капсула - это набор нейронов, которые индивидуально активируются для различных свойств типа объекта, таких как положение, размер и оттенок. Формально капсула - это набор нейронов, которые вместе создают вектор активности с одним элементом для каждого нейрона, чтобы удерживать значение экземпляра этого нейрона (например, оттенок). Графические программы

используют значение создания экземпляра для рисования объекта. CapsNets пытается извлечь их из своих входных данных. Вероятность присутствия объекта на конкретном входе - это длина вектора, а ориентация вектора количественно определяет свойства капсулы.

Искусственные нейроны традиционно выводят скалярную активацию с действительным знаком, которая в общих чертах представляет вероятность наблюдения. Capsnets заменяют детекторы функций скалярного вывода капсулами с векторным выводом, а max-pooling - маршрутизацией по соглашению.

Поскольку капсулы независимы, когда несколько капсул совпадают, вероятность правильного обнаружения намного выше. Минимальный кластер из двух капсул с учетом шестимерного объекта согласуется с точностью до 10% случайно только один раз из миллиона испытаний. По мере увеличения количества измерений вероятность случайного совпадения в более крупном кластере с более высокими измерениями экспоненциально уменьшается.

Капсулы на более высоких уровнях принимают выходные данные от капсул на более низких уровнях и принимают те, выходы которых группируются. Кластер заставляет высшую капсулу выводить высокую вероятность наблюдения присутствия сущности.

Таким образом, предложенный Хинтоном новый вид сети старается исправить эти недостатки следующим образом:

1. Замена нейронов капсулами. Конструкция капсулы строится на устройстве искусственного нейрона, но расширяет его до векторной формы, чтобы обеспечить более мощные репрезентативные возможности. Также вводятся весовые коэффициенты матрицы для кодирования иерархических связей между особенностями разных слоев. Достигается эквивариантность нейронной активности в отношении изменений входных данных и инвариантности в вероятностях обнаружения признаков. «Выход нейрона – вектор, способный передать позу объекта».
2. Замена слоя пулинга на алгоритм маршрутизации по соглашению. В результате происходит не простой выбор максимальных значений, а инкапсуляция ценной информации в векторе выхода.

Базовая капсульная архитектура

Базовая капсульная архитектура для решения задачи классификации CapsNet была предложена в 2017 состояла из двух сверточных и одного полносвязного капсульного слоя (рис. 4).

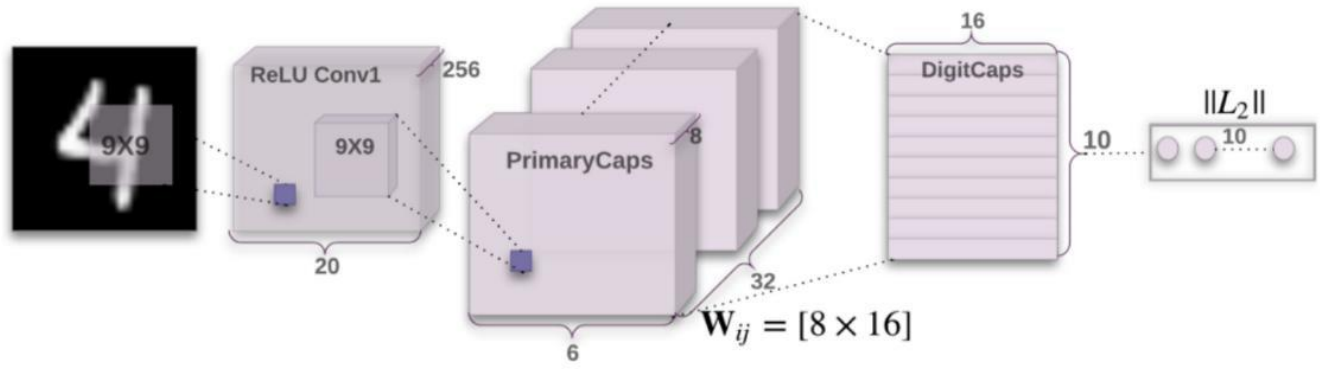

Рис. 4. Архитектура CapsNet

Для понимания работы сети CapsNet введем пример входного изображения (рис. 5).

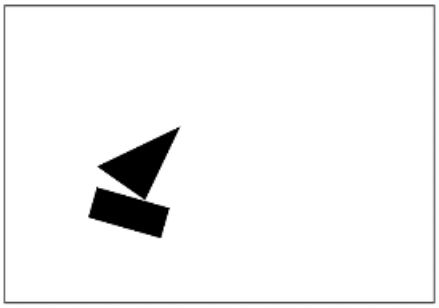

Рис. 5. Пример входного изображения

В первом слое (ReLU Conv1) на входном изображении применялась операция свертки с размером ядра 9 × 9 и 256 каналами. Шаг окна равнялся 1 пикселю. После чего применялась нелинейная функция активации ReLU

Второй слой (PrimaryCaps) обладает теми же параметрами, что и первый, за исключением смещения окна, которое сдвигалось на два пикселя. Группы капсул образовывались путем деления 256 каналов на 32 части по 8 значений в каждой. Таким образом, для изображения с одним цветовым каналом и размером 28 пикселей количество капсул равняется 1152. Для каждой капсулы применяется нелинейная функция активации, которую авторы назвали функцией сжатия (англ. squash function). В ней все векторы, которые имеют короткую длину, «сжимают» к длине, близкой к нулю. А векторы с большой длиной до длины, близкой к единице:

$$\bar{v}_j = \frac{\|\bar{s}_j\|^2}{1 + \|\bar{s}_j\|^2} \frac{\bar{s}_j}{\|\bar{s}_j\|} \quad (1)$$

где s_j – входное векторное значение капсулы j.

Правая часть уравнения (синий прямоугольник) масштабирует входной вектор так, что вектор будет иметь длину блока, а левая сторона (красный

прямоугольник) выполняет дополнительное масштабирование. Суть функции в том, что длина вектора – это уверенность нейронной сети в существовании признака. Это позволяет акцентировать внимание только на существующих признаках объекта. Единица – это максимальное значение длины вектора и полная уверенность в существовании признака. В примере получено две капсулы с наибольшей величиной: признака треугольника и прямоугольника (рис. 6).

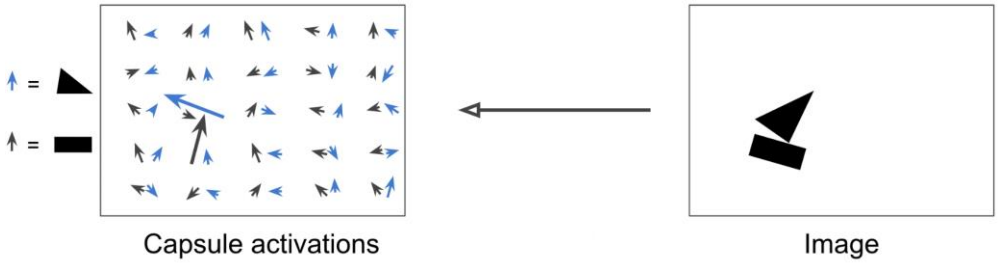

Рис. 6. Длина капсул отражает уверенность в существовании признака объекта

В последнем, полносвязном слое (DigitCaps) происходят процедуры согласованной и динамической маршрутизации (англ. routing by agreement and dynamic routing). Выходное значение $\hat{u}_j$ слоя DigitCaps – это произведение всех векторов, полученных на предыдущем слое i, путем умножения значений на матрицу весов W_{ij}:

$$\hat{u}_j = \sum_i W_{\text{ij}} \bar{u}_i \qquad (2)$$

Предположим, в тестовом примере сеть определяет два класса: дома и лодки. В таком случае слой DigitCaps будет состоять из двух капсул, каждая из которых примет выход всех значений капсул: прямоугольник и треугольник из слоя PrimaryCaps. После применения формул (2) и (1), длина векторного выхода капсулы класса «лодка» будет ближе к единице, а капсулы класса «дом» – к нулю (рис. 7). Данную процедуру авторы назвали процедурой согласованной маршрутизации.

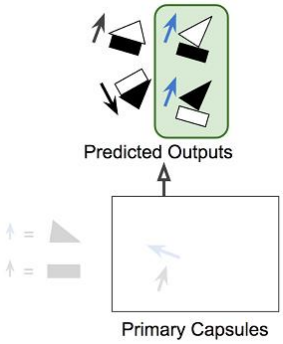

Рис. 7. Принцип работы согласованной маршрутизации

Далее начинается процедура динамической маршрутизации. В начале алгоритма входные векторные значения капсул представляются в качестве

элементов векторного пространства с равным весом и центром масс (рис. 8).

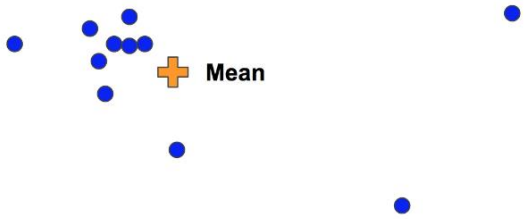

Рис. 8. Пример в двумерном пространстве

На основе c_{ij} – коэффициентов, полученных при помощи функции маршрутизированного мягкого максимума (англ. routing softmax function) и вычисляемых с помощью (3), возможно получить $\bar{s}_j$, который является взвешенной суммой всех векторов $\hat{u}_j$ (4).

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{3}$$

где b_{ij} – нулевой тензор, но в процессе итераций динамической маршрутизации принимающий значение $b_{ij} = b_{ij} + \hat{u}_j v_j$.

$$\bar{s}_j = \sum_i c_{ij} \hat{u}_j \tag{4}$$

Из-за различия коэффициентов веса, элементы векторного пространства будут обладать другой точкой центра масс (рис. 9). Данную процедуру можно повторить множество раз. Однако, стоит учесть, что это вычислительно сложная операция. Выходом слоя DigitCaps являются вектора $\bar{v}_j$ после r итераций.

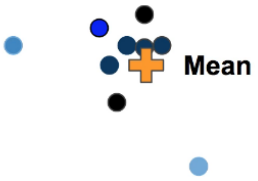

Рис. 9. Пример в двумерном пространстве

Принятие решения о классе объекта происходит путем сравнения размеров выходов капсул по L_2 норме. Предсказанный класс обладает наибольшим значением.

Чтобы избежать переобучения сети, которое возникает сравнительно быстро, необходим метод регуляризации для капсульных сетей – декодер (рис. 10). Он состоит из двух полносвязных слоев с функцией активации ReLU и одного полносвязного слоя с сигмоидальной функцией активацией. На вход подается матрица, полученная из последнего слоя архитектуры. Входные значения матрицы капсул приравниваются к нулю, кроме одной строки, которая имеет наибольшую L_2 норму. Выходом декодера является вектор изображения, эквивалентный подаваемому на вход в архитектуре CapsNet. Полученный вектор подается в функцию потерь декодера, которая может найти разницу между

полученным и исходным изображением. Например, это может быть функция среднеквадратической ошибки.

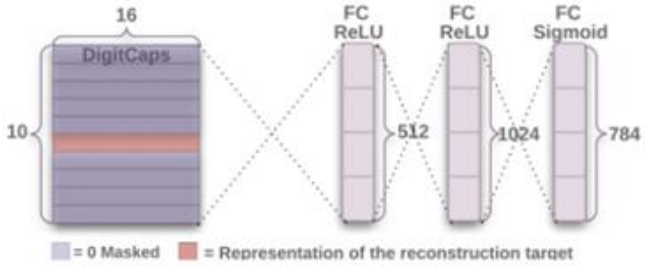

Рис. 10. Декодер CapsNet

Для обучения сети требуется функция ошибки, которая будет максимизировать входные векторные значения капсулы верного класса, минимизируя остальные:

$$L_k = T_k max(0, m^+ - ||v_k||)^2 + \lambda(1 - T_k)\, max(0, ||v_k|| - m^-)^2 \qquad (5)$$

где:

– T_k – компонента унитарного кода, которая равна единице, если k–верный номер класса;

– m^+ – константа длины для входного значения капсулы верного класса. Рекомендуемое для неё значение $= 0.9$;

– $||v_k||$ – L_2 норма выходной капсулы k;

– m^- – константа длины для входных значений капсул других классов. Рекомендуемое для неё значение $= 0.1$;

– λ – коэффициент для предотвращения сокращения длин векторных значений всех капсул на начальных этапах обучения. Рекомендуемое значение $= 0.5$.

Итоговое значение функции потерь равна сумме ошибок для всех входных значений капсул и функции потерь декодера.

Глубокие капсульные нейронные сети (DeepCaps)

Из-за неглубокой архитектуры CapsNet нельзя получить хорошие показания точности на наборе данных сложнее чем MNIST (объёмная база данных образцов рукописного написания цифр), но добавить больше слоев в изначальную сеть или увеличить количество итераций в динамическую маршрутизацию не представляется возможным по двум причинам:

1) из-за полносвязной архитектуры с увеличением количества капсул возникает ситуация затухания градиента;

2) вычислительная сложность операции динамической маршрутизации, которая существенно увеличит время и необходимые мощности для обучения и использования сети.

Чтобы обойти данные ограничения, была предложена глубокая капсульная архитектура с независящим от класса декодером.

Для описания работы данной нейросети потребуется ввести обозначения:

– w^l – ширина и высота карты признаков слоя l,

– n^l – размер капсулы слоя l,

– c^l – количество капсул слоя l,

– Φ^l – тензор слоя l

В DeepCaps содержится 15 сверточных слоев – ConvCaps, которые были изменены для работы с капсулами. Данные слои принимают на вход четырхмерный тензор вида $\Phi^l \in \mathrm{R}^{(w^l, w^l, c^l, n^l)}$ и применяют операцию свертки на преобразованном тензоре $\Psi^l \in \mathrm{R}^{(w^l, w^l, c^l \times n^l)}$. Далее полученный тензор разбивается на c^{l+1} частей размера n^{l+1} и применяется измененная капсульная функция (формула 1) активации сквош 3D:

$$\hat{s}_{pqr} = \frac{\left\|\bar{s}_{pqr}\right\|^2}{1 + \left\|\bar{s}_{pqr}\right\|^2} \times \frac{\bar{s}_{pqr}}{\left\|\bar{s}_{pqr}\right\|}$$

В слое, который авторы назвали ConvCaps3D, используется функция 3D свертки и 3 итерации пространственной динамической маршрутизации (рис. 11).

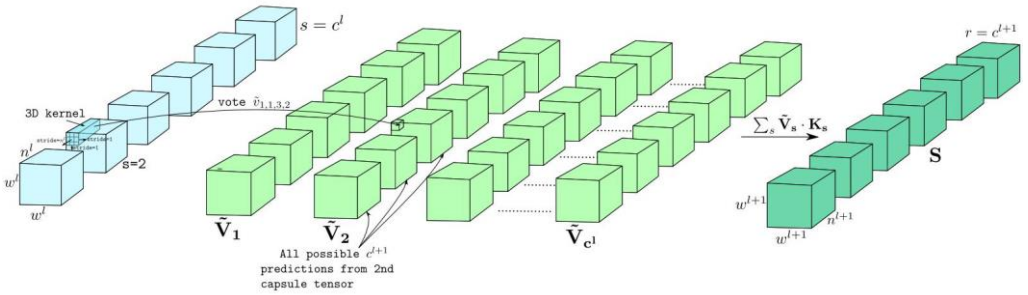

Рис. 11. Схема работы слоя ConvCaps3D

Для его реализации, требуется улучшить формулу (3) для работы с трехмерным пространством:

$$k_{pqrs} = \frac{exp(b_{pqrs})}{\sum_x \sum_y \sum_z exp(b_{xyzs})} \tag{7}$$

где $p, q \in w^{l+1}, r \in c^{l+1}, s \in c^l$.

Алгоритм слоя ConvCaps3D получает на вход тензор $\Phi^l \in R(w^l, w^l, c^l, n^l)$, количество итераций i, c^{l+1} – количество капсул слоя $l+1$ и n^{l+1} – размер выходного вектора капсулы слоя $l+1$. При помощи операции изменения размерности тензора, $\Phi^l \in R(w^l, w^l, c^l, n^l)$ преобразуется в тензор $\widehat{\Phi}^l \in R(w^l, w^l, c^l, n^l)$ к которому применяется трехмерная операция свертки, в результате применения которой получаем тензор $V \in R(w^{l+1}, w^{l+1} c^l c^{l+1} \times n^{l+1})$. Далее полученный тензор проходит через операцию изменения

размерности тензора, выходом которой является тензор $\hat{V} \in R(w^{l+1}, w^{l+1}, n^{l+1}, c^{l+1}, c^{l})$. Следующим шагом является инициализация нулевого тензора $B \in R(w^{l+1}, w^{l+1}, c^{l+1}, c^{l})$. Пусть $p, q \in w^{l+1}, r \in c^{l+1}, s \in c^{l}$, тогда для i итераций:

1) для всех p, q, r в тензоре B применяется формула (7) для вычисления коэффициентов k_{pqrs};

2) получение взвешенного значения капсулы: $S_{\mathrm{pqr}} = \sum_s k_{\mathrm{kqrs}} \hat{V}_{\mathrm{pqrs}}$;

3) применение трехмерной функции сжатия (6) на взвешенные значения капсул S_{pqr} для получения нормированных значений капсул $\hat{S}_{\mathrm{pqr}}$;

4) обновление коэффициентов: $b_{pqrs} = b_{pqrs} + \hat{S}_{pqr} \cdot \hat{V}_{pqrs}$.

Выходом слоя ConvCaps3D является тензор $\hat{S}_{pqr}$ после i итераций.

Слой FlatCaps изменяет размерность тензора на двумерную матрицу размера $(w_l \times w_l \times c_l, n_l)$.

Работа слоя FCCaps похожа на работу обычного полносвязного слоя. Каждый выход капсул умножается на матрицу весов $W_{ij} \in R^{(n^l, n^{l+1})}$.

1.3.2. Архитектура сети

Архитектура сети представлена на рисунке 12.

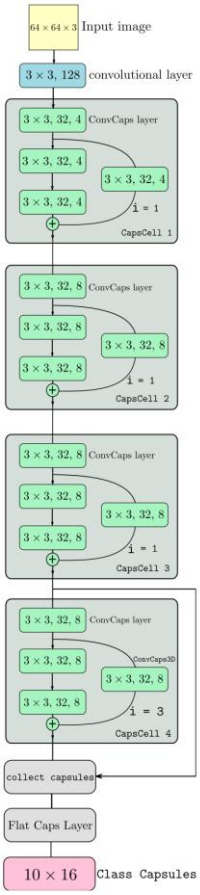

Рис. 12. Модель DeepCaps

Первые три блока сети состоят из четырех слоев ConvCaps. Один из слоев в каждом блоке реализован в остаточном соединении. Третий блок соединен непосредственно со слоем сбора капсул и конкатенируется с последним блоком в слое сбора капсул. Четвертый блок использует слой ConvCaps3D в остаточном соединении с тремя итерациями пространственной динамической маршрутизации. Следующий шаг – конкатенация выхода капсул с третьего и последнего блока, предварительно преобразованного в “плоский” вид (слой ”collect capsules”). Далее, после сбора капсул слоем ”collect capsules”, они попадают в полносвязный капсульный слой Flat Caps Layer. Полученные значения капсул можно задействовать в декодере для регуляризации сети. Функция потерь совпадает с функцией ошибки CapsNet (формула 5)