

Машинное обучение

Лекция 4.

Глубокие нейронные сети.

Часть 2.

Виды сверточных нейронных сетей

LeNet. Первая успешная реализация свёрточной нейронной сети была разработана Yann LeCun в 1990. Из них самой известной стала архитектура LeNet, которая использовалась для чтения ZIP-кодов, цифры и др.

AlexNet. Первая работа, которая популяризовала свёрточные нейронные сети в компьютерном зрении, разработана Alex Krizhevsky, Ilya Sutskever и Geoff Hinton. AlexNet была протестирована на соревновании ImageNet ILSVRC в 2012 году и значительно опередила конкурента на втором месте (процент ошибок: 16% против 26%). Архитектура сети была очень похожа на архитектуру LeNet, но была глубже, больше и использовала последовательности свёрточных слоёв (до этого было стандартной практикой использовать только комбинацию свёрточного слоя сразу за которым следовал слой подвыборки).

Виды сверточных нейронных сетей (2)

ZFNet. Победителем ILSVRC 2013 года стала свёрточная нейронная сеть от Matthew Zeiler и Rob Fergus. Стала она известна под названием ZFNet. Это была усовершенствованная версия AlexNet, которая использовала другие значения гипер-параметров, в частности увеличивавшая размер среднего свёрточного слоя и уменьшавшая шаг и размер фильтра на первом свёрточном слое.

GoogLeNet. Победителем ILSVRC 2014 года стала свёрточная нейронная сеть от Szegedy et al. из Google. Основным изменением в архитектура стал Inception-модуль, который позволял значительно снизить количество параметров в сети (4 Мб против 60 Мб в AlexNet). В добавок к этому сеть использует подвыборку по среднему значению вместо полносвязных слоёв к концу сети, исключая тем самым большое количество параметров, которые не имеют большого значения. У данной версии сети есть несколько модификаций, одна из последних — Inveption-v4.

Виды сверточных нейронных сетей (3)

VGGNet. Вслед за победителем 2014 ILSVRC была сеть от Karen Simonyan и Andrew Zisserman, которая стала так же известна как VGGNet. Основным вкладом этой нейронной сети в развития представлений о свёрточных нейронных сетях был тот факт, что глубина сети является ключевым компонентом хорошей производительности. Финальная версия их сети содержала 16 пар свёрточных + полносвязных слоёв и использовала единые размеры фильтров (3x3 для свёртки и 2x2 для операции подвыборки). Их предобученная модель находится в публичном доступе и с ней можно поэкспериментировать. Обратная сторона медали VGGNet — стоимость вычислений и использование большого количества памяти (140Мб). Большинство из этих параметров находятся на первом полносвязном слое и, как было в дальнейшем изучено и протестировано, эти полносвязные слои могут быть исключены без влияния на производительность, но с колоссальным уменьшением количества требуемых параметров.

ResNet. Residual-сеть была разработана Kaiming He et al. и стала победителем в ILSVRC 2015. Она активно использует пропуск связей и блоковую нормализацию. В архитектуре так же отсутствуют полносвязные слои в конце сети. На данный момент это лучшее решение с использованием свёрточных нейронных сетей (на май 2016).

Достижения разных глубоких сетей при распознавании реальных объектов

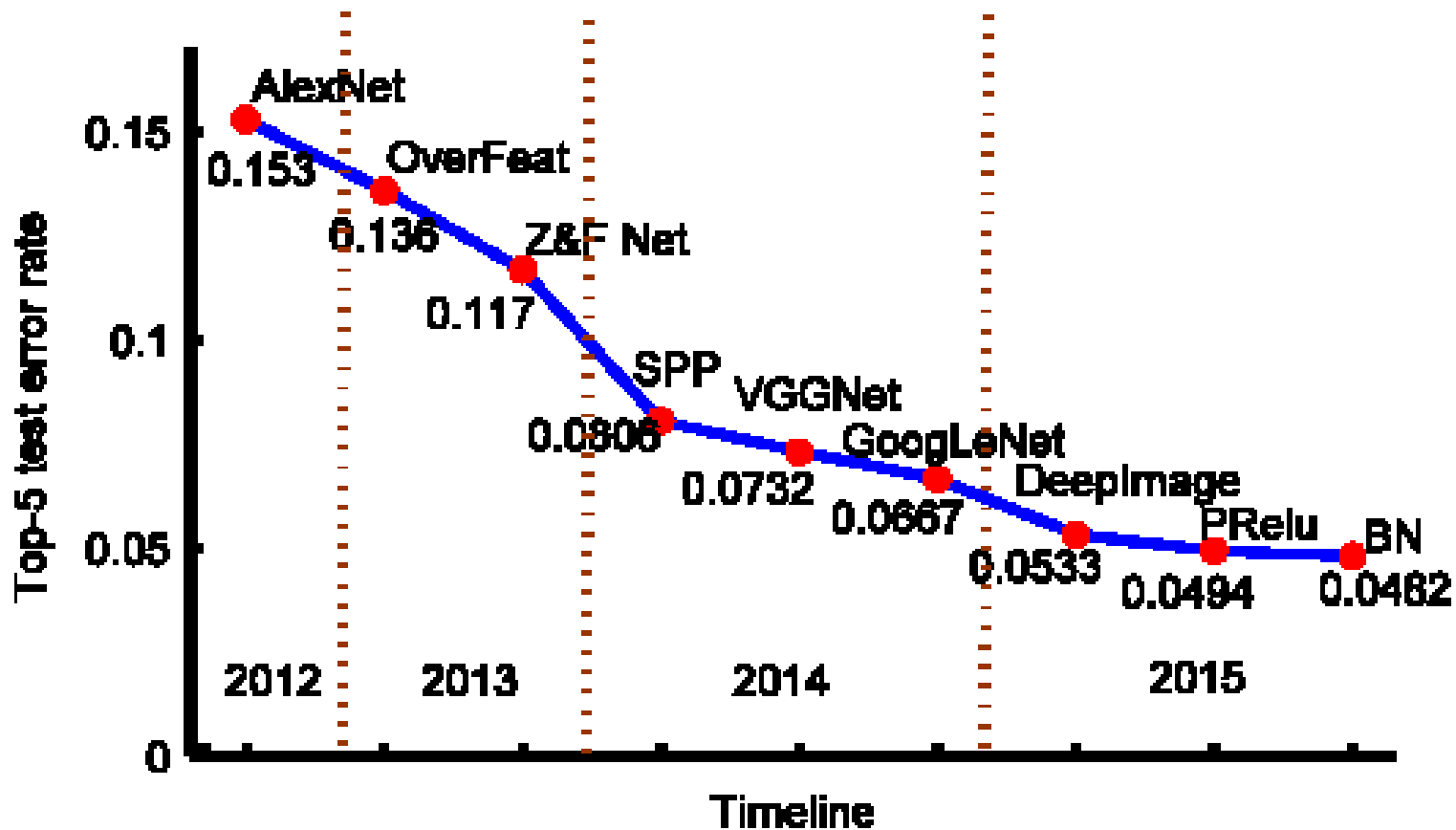
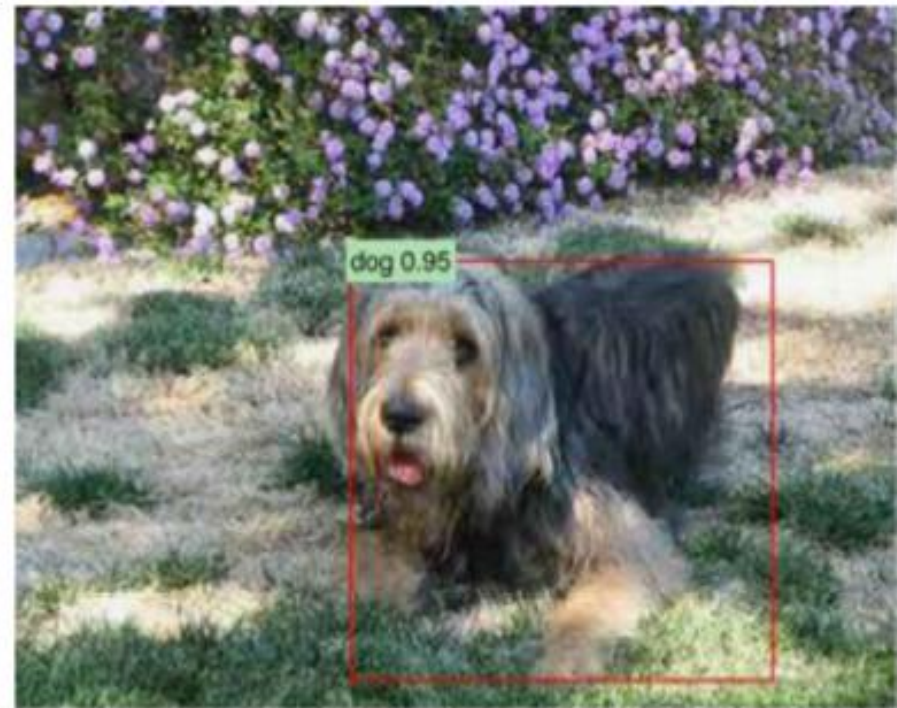
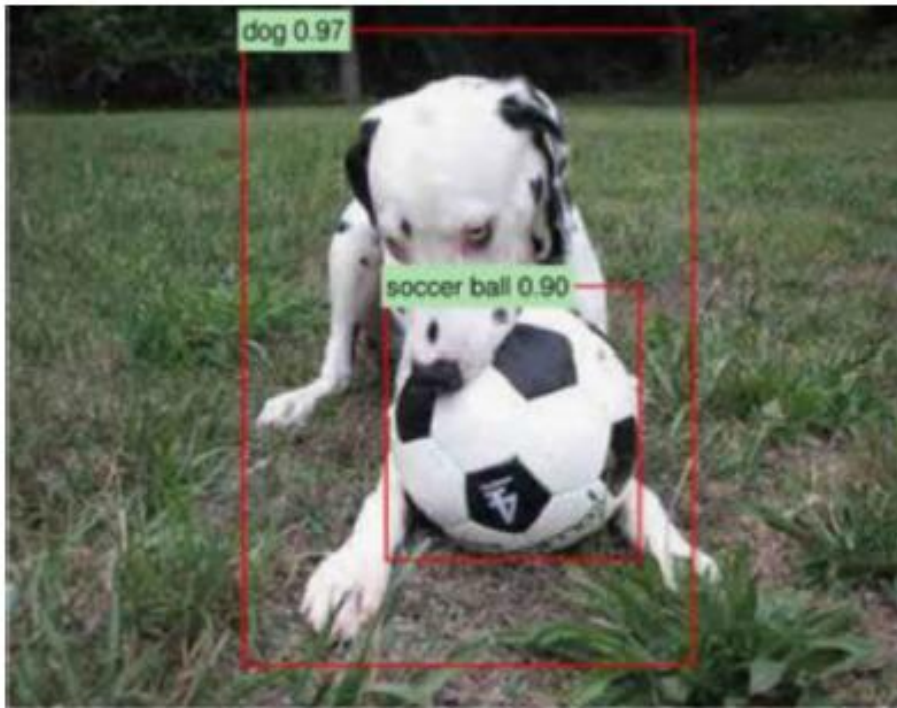
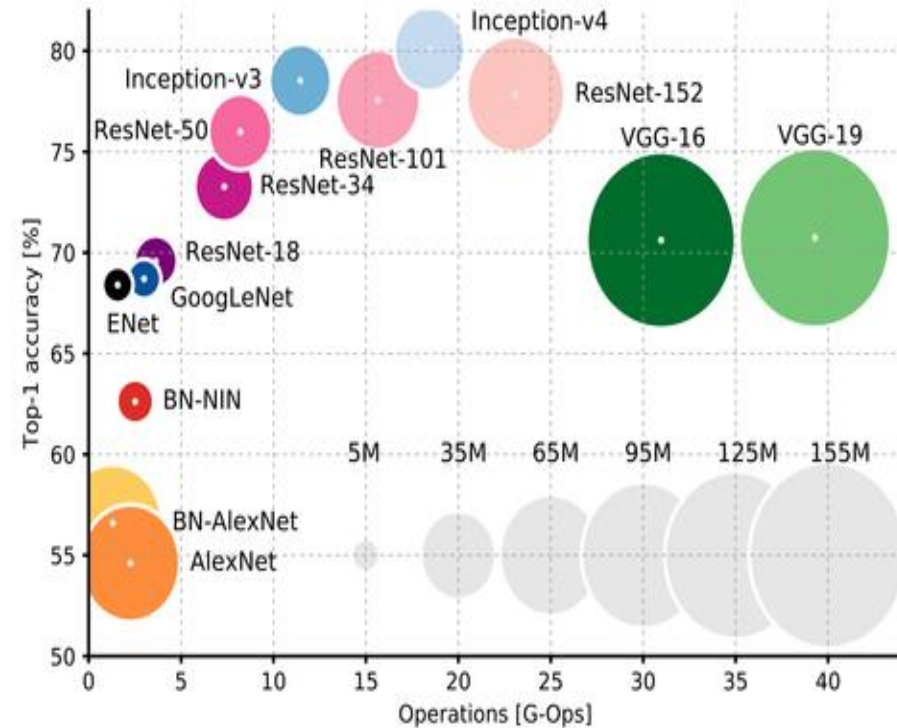
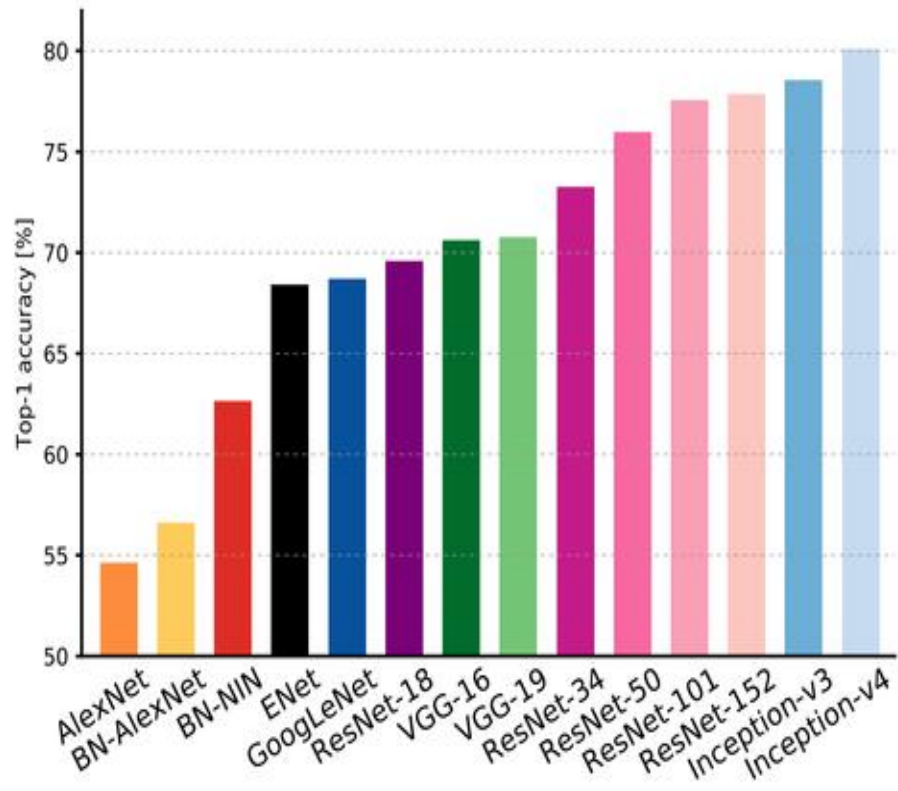


Fig. 18. ImageNet classification results on test dataset.

Достижения разных глубоких сетей при распознавании реальных объектов



Сравнение разных сверточных сетей



Библиотеки для deep learning

#	Название	Язык	ОС	FC NN	CNN	AE	RBM
1	DeepLearnToolbox	Matlab	Windows, Linux	+	+	+	+
2	Theano + Keras Tensorflow + Keras	Python	Windows, Linux, Mac	+	+	+	+
3	Pylearn2	Python	Linux, Vagrant	+	+	+	+
4	Deepnet	Python	Linux	+	+	+	+
5	Deepmat	Matlab	?	+	+	+	+
6	Torch	Lua, C	Linux, Mac OS X, iOS, Android	+	+	+	+
7	Darch	R	Windows, Linux	+	—	+	+
8	Caffe	C++, Python, Matlab	Linux, OS X	+	+	—	—
9	nnForge	C++	Linux	+	+	—	—
10	CXXNET	C++	Linux	+	+	—	—
11	Cuda-convnet	C++	Linux, Windows	+	+	—	—
12	Cuda CNN	Matlab	Linux, Windows	+	+	—	—

Сокращения в таблице

- FC NN - fully connected neural network (полносвязные НС),
- CNN - convolutional neural network (сверточные НС)
- AE - autoencoder (автокодировщики),
- RBM - restricted Boltzmann machine (ограниченные машины Больцмана)

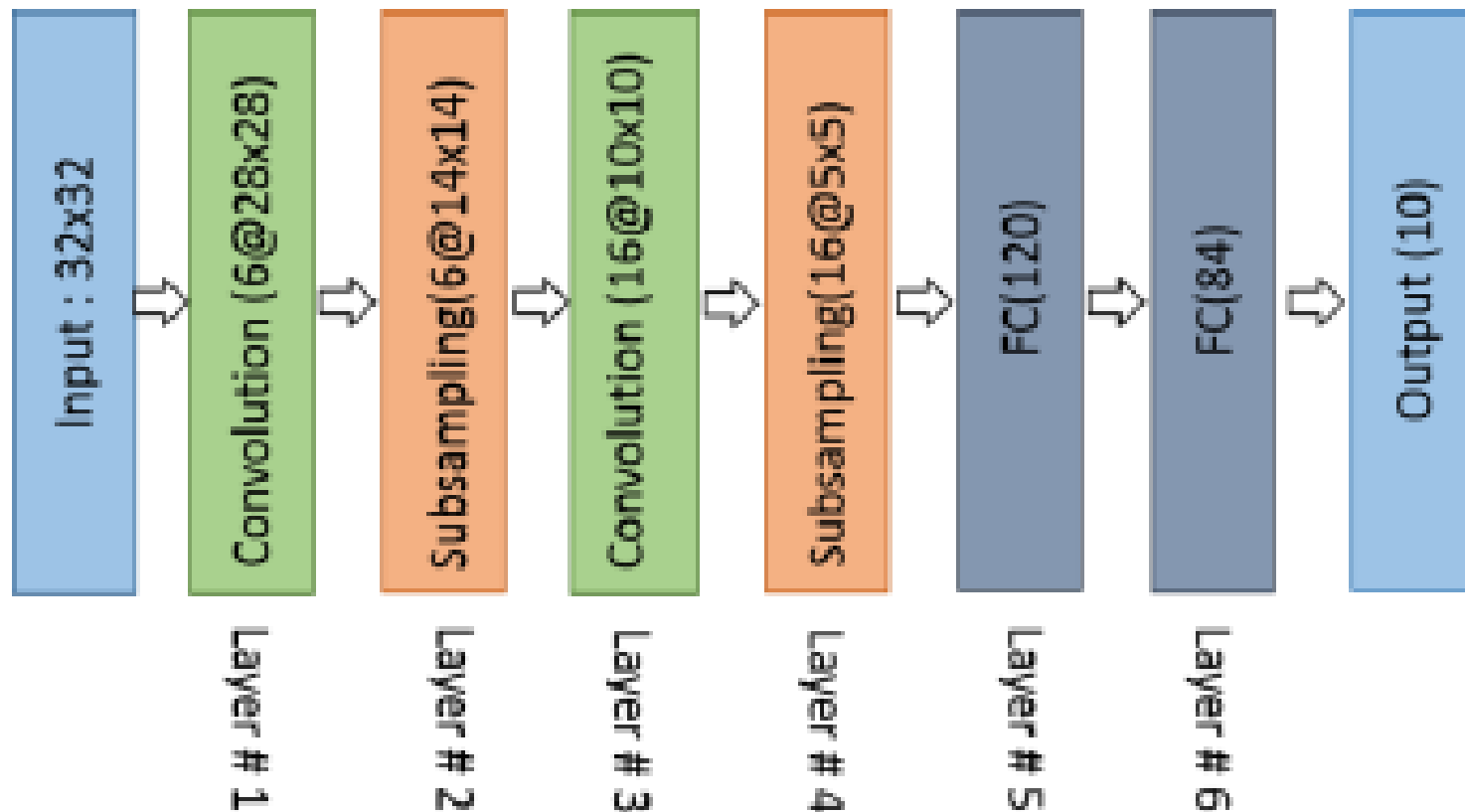
Типы сверточных сетей

LeNet

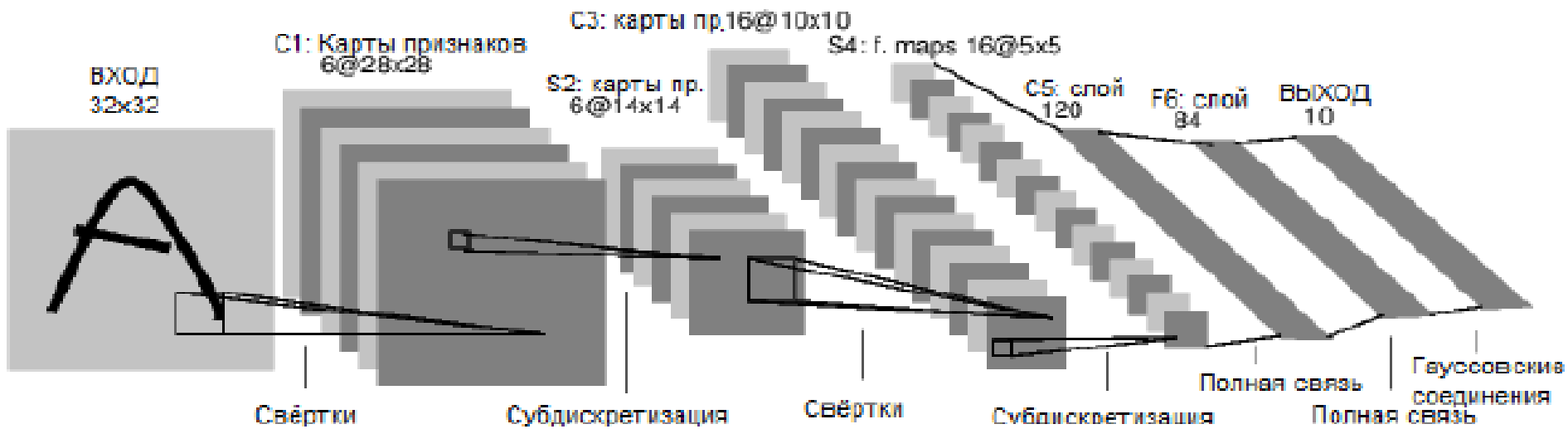
Хотя LeNet была предложена в 1990-ых годах, ограниченные возможности вычислений и объем памяти затрудняли реализацию алгоритма до 2010 года. Тем не менее, Ян Лекусн предложил CNN с алгоритмом обратного распространения и экспериментировал с набором данных рукописных цифр для достижения самых современных показателей точности.

Его архитектура хорошо известна как LeNet-5.

Базовая конфигурация LeNet-5: 2 сверточных (conv) слоя, 2 слоя подвыборки, 2 полносвязных уровня и выходной уровень с гауссовским соединением. Общее количество весов и вычислений MAC составляет 431k. и 2,3M. соответственно.



LeNet-5



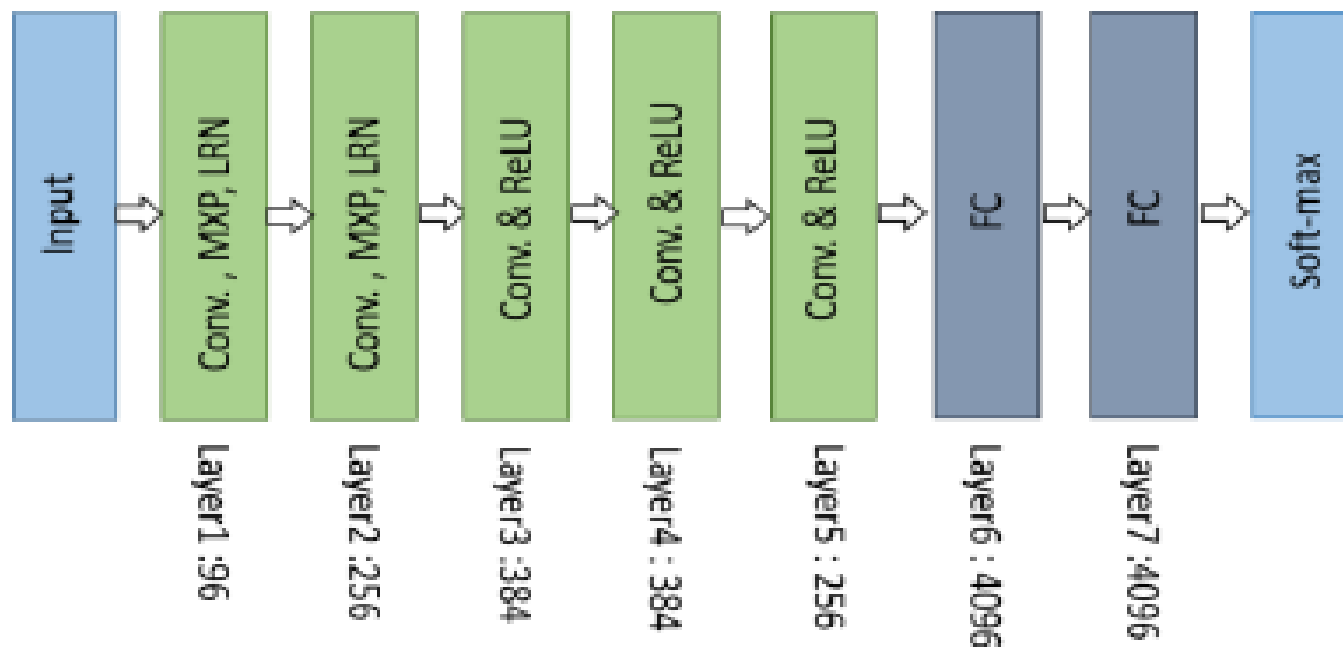
AlexNet

В 2012 году Алекс Крижевский и другие предложили более глубокую и более широкую модель CNN по сравнению с LeNet и выиграли сложнейшую задачу ImageNet для визуального распознавания объектов, названную ImageNet Large Scale Visual Recognition Challenge (ILSVRC) в 2012 году.

AlexNet достигла самой совершенной точности распознавания во всех традиционных подходах к компьютерному обучению и компьютерному видению.

Первый сверточный слой выполняет свертку и пулинг с локальной нормализацией отклика (LRN), где используются 96 различных фильтров размером 11×11 .

Пулинг операция выполняется с 3×3 фильтрами с размером шага 2. Те же операции выполняются на втором уровне с фильтрами 5×5 . Фильтры 3×3 используются в третьем, четвертом и пятом сверточных слоях с 384, 384 и 296 картами соответственно и функцией активации ReLU, что позволило снизить количество эпох обучения в шесть раз. Два полносвязных слоя используются с новой функцией отсева (dropout), за которым следует слой soft-max в конце. Общее количество весов и вычислений MAC для всей сети составляют соответственно 61M и 724M

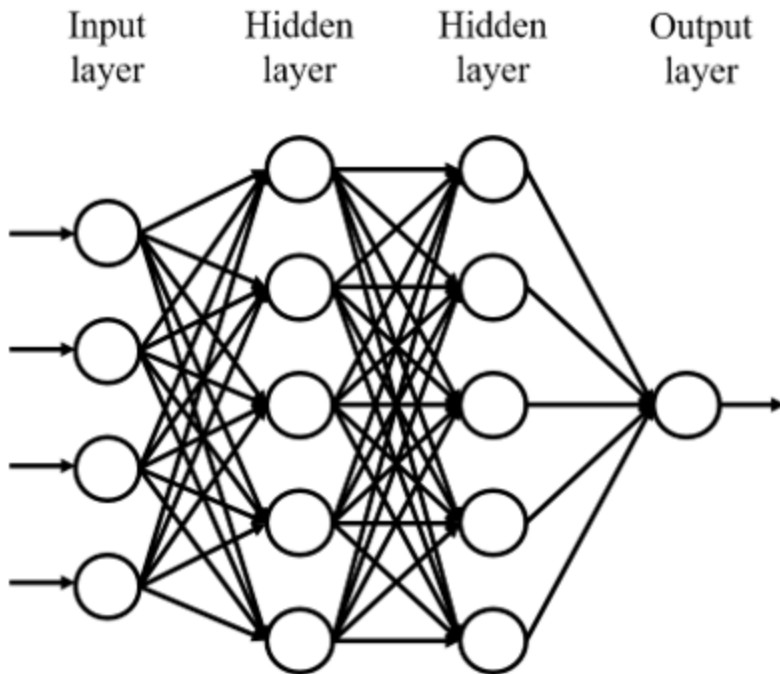


LRN нормализует область вокруг возбужденного нейрона-признака. Он становится более чувствительным. После ReLU его значение может быть слишком большим, LRN его уменьшает.

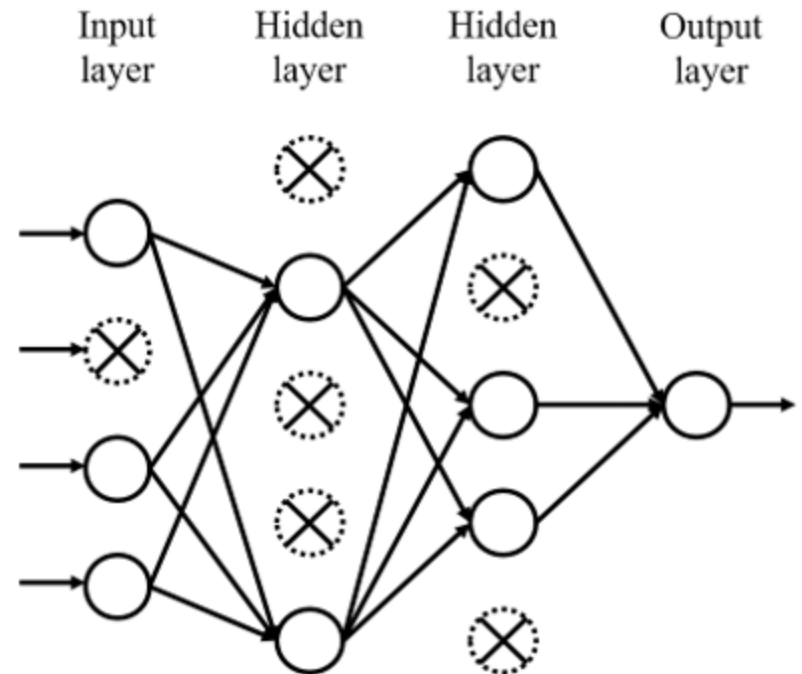
Примененная техника отсева (Dropout). Она заключается в случайном отключении каждого нейрона на заданном слое с вероятностью p на каждой эпохе. После обучения сети, на стадии распознавания, веса слоёв, к которым был применён dropout, должны быть умножены на $1/p$. Dropout выступает в роли регуляризатора, не позволяя сети переобучаться.

В этой сети впервые был применен GPU.

Dropout



Unthinned network before applying dropout



Thinned network after applying dropout

ZFNet

В 2013 году Мэтью Зейлер и Роб Ферг выиграли ILSVRC 2013 года с архитектурой CNN, которая была продолжением AlexNet. Сеть называлась ZFNet. Поскольку CNN являются требовательными к вычислительным методам, то оптимальное использование параметров требуется с точки зрения сложности модели.

Архитектура ZFNet – это усовершенствование AlexNet, разработанное путем настройки с сетевыми параметрами последнего. ZFNet использует ядра 7×7 вместо ядер 11×11 , чтобы значительно уменьшить количество весов. Это значительно уменьшает количество сетевых параметров и улучшает общую точность распознавания.

Nin

В этой модели вводится пара новых понятий.

Первая концепция заключается в использовании многослойной свертки, где свертки выполняются с фильтром 1×1 , что помогает добавить в модели больше нелинейности. Это помогает увеличить глубину сети, которая затем может быть упорядочена с отсевом. Эта концепция часто используется в bottleneck слое модели глубокого обучения.

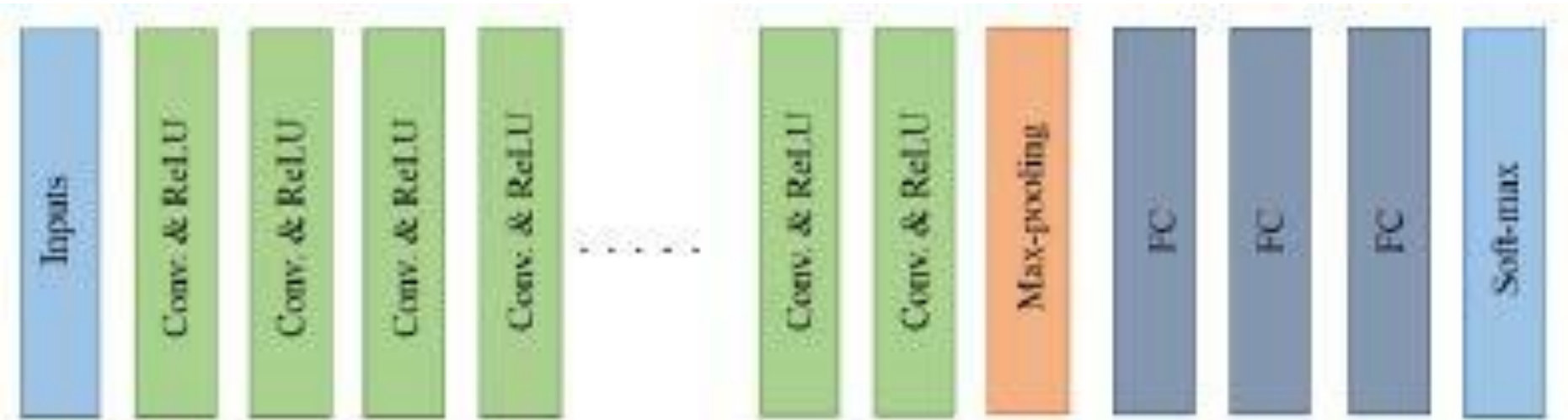
Вторая концепция – использовать Global Average Pooling (GAP) как альтернативу полносвязанных слоев. Это помогает значительно уменьшить количество сетевых параметров. GAP значительно меняет структуру сети. Применяя GAP на большой карте объектов, мы можем генерировать конечный низкоразмерный вектор-объект, не уменьшая размерности карт признаков.

VGGNet

Группа визуальных геометрий (VGG) была лидером ILSVRC 2014 года. Основной вклад этой работы заключается в том, что она показывает, что глубина сети является важным компонентом для достижения лучшего признания или точности классификации в CNN.

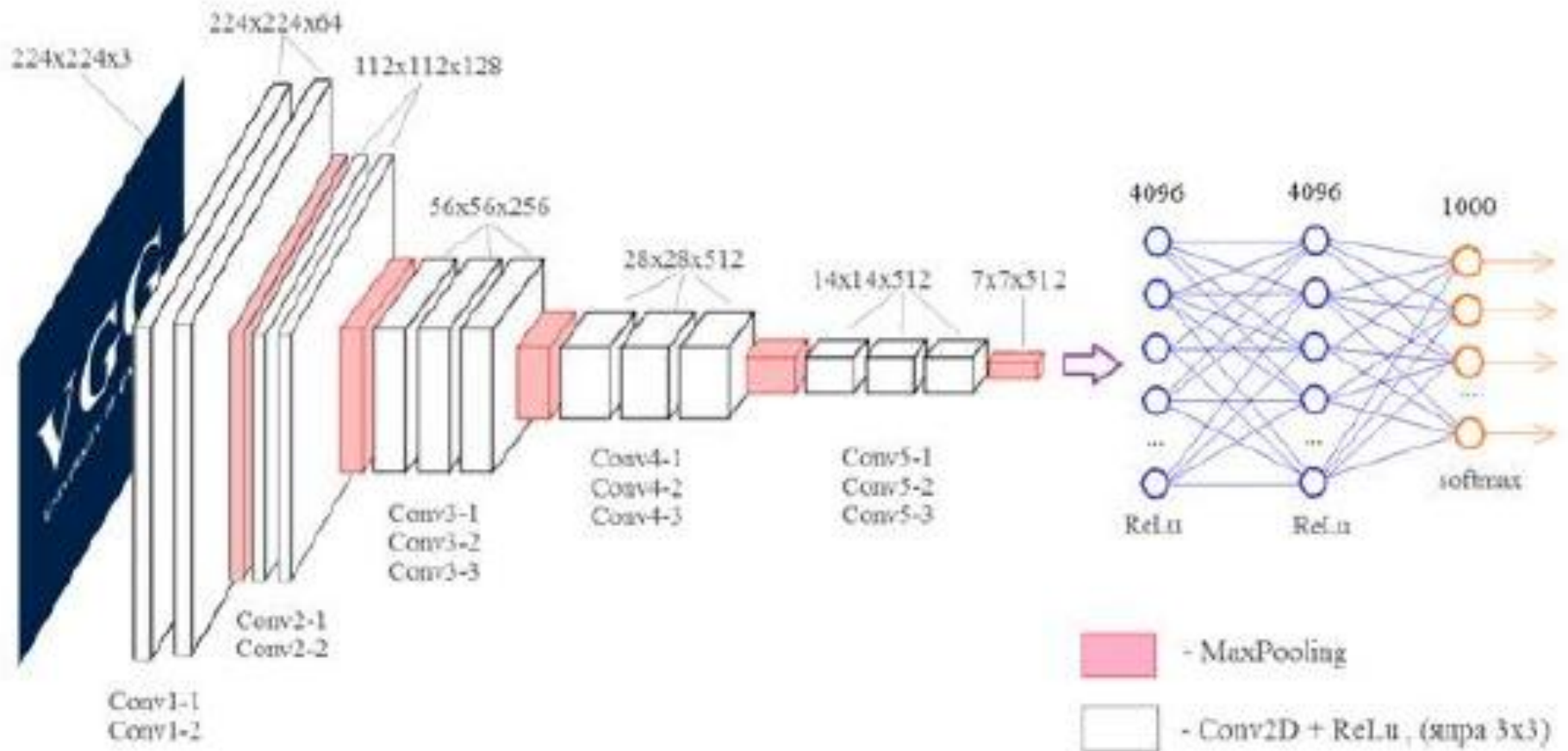
Архитектура VGG состоит из двух сверточных слоев, оба из которых используют функцию активации ReLU. После активационной функции сеть представляет собой один уровень пулинга и несколько полносвязных слоев, также используя функцию активации ReLU.

Последним слоем модели является слой soft-max для классификации. В VGG-E размер фильтра свертки изменяется на фильтр 3x3 с шагом 2. Три модели VGG-E, VGG-11, VGG-16 и VGG-19; были предложены модели 11, 16 и 19 слоев соответственно.



Все версии моделей VGG-E закончили одинаково с тремя полносвязными слоями. Однако количество слоев свертки варьировалось, VGG-11 содержало 8 слоев свертки, VGG-16 имело 13 слоев свертывания, а VGG-19 имело 16 слоев свертки. VGG-19, самая вычислительно требовательная модель, содержала 138M весов и имела вычислений MAC 15,5G.

Структура VGG-16



VGG-16	VGG-19
input (224x244x3)	
conv3-64 conv3-64	conv3-64 conv3-64
maxpool	
conv3-128 conv3-128	conv3-128 conv3-128
maxpool	
conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
maxpool	
conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool	
conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
maxpool	
FC-4096	
FC-4096	
FC-1000	
softmax	

Реализация VGG-16 в Keras

```
tf.keras.applications.VGG16(  
    include_top=True,  
    weights="imagenet",  
    input_tensor=None,  
    input_shape=None,  
    pooling=None,  
    classes=1000,  
    classifier_activation="softmax",  
)
```

`include_top` – использовать ли полносвязные слои (True – да; False – нет);

`weights` – определяет значения весов (по умолчанию берутся обученные на коллекции изображений ImageNet);

`classes = 1000` – число выходных нейронов;

`classifier_activation` – функция активации для выходных нейронов.

Использование VGG-16

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from google.colab import files
from io import BytesIO
from PIL import Image
model = keras.applications.VGG16()
uploaded = files.upload()
img = Image.open(BytesIO(uploaded['ex224.jpg']))
plt.imshow( img )
# приводим к входному формату VGG-сети img = np.array(img)
x = keras.applications.vgg16.preprocess_input(img)
print(x.shape)
x = np.expand_dims(x, axis=0)
# прогоняем через сеть
res = model.predict( x )
print(np.argmax(res))
```

GoogLe LeNet

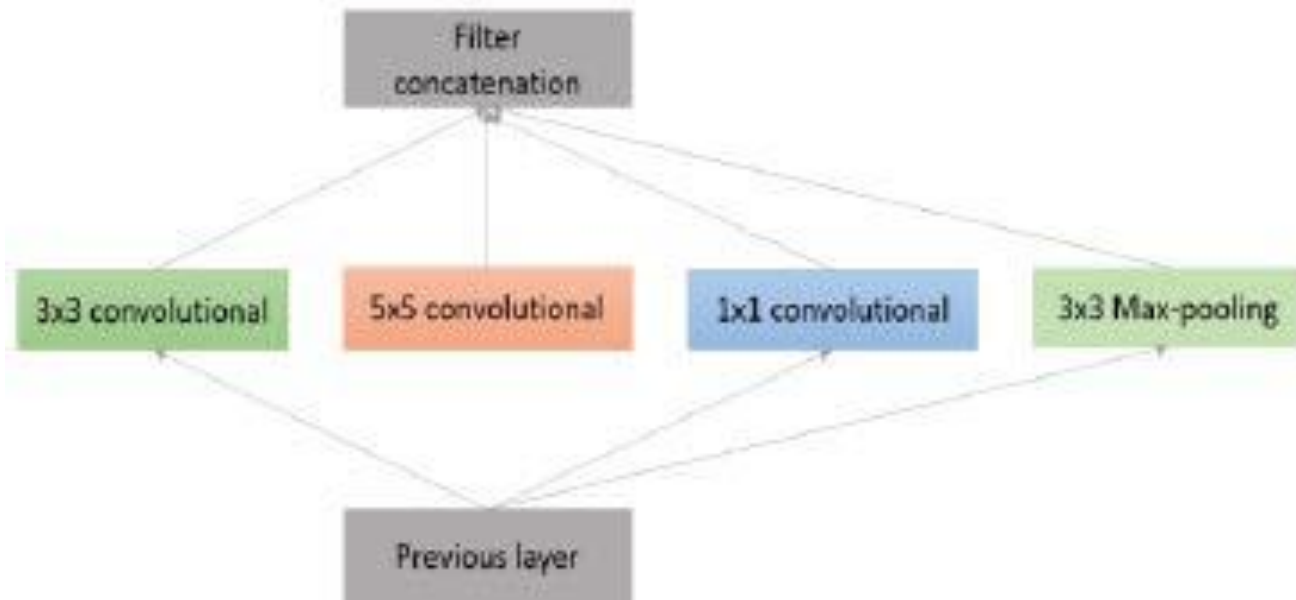
GoogLeNet, победитель ILSVRC 2014, была моделью, предложенной Кристианом Сегеди от Google с целью снижения сложности вычислений по сравнению с традиционным CNN.

Предлагаемый метод состоял в том, чтобы включить «Inception слои», которые имели переменные восприимчивые поля, которые были созданы с разными размерами ядер. Эти восприимчивые поля создавали операции, которые фиксировали разреженные корреляционные шаблоны в новом стеке карт объектов.

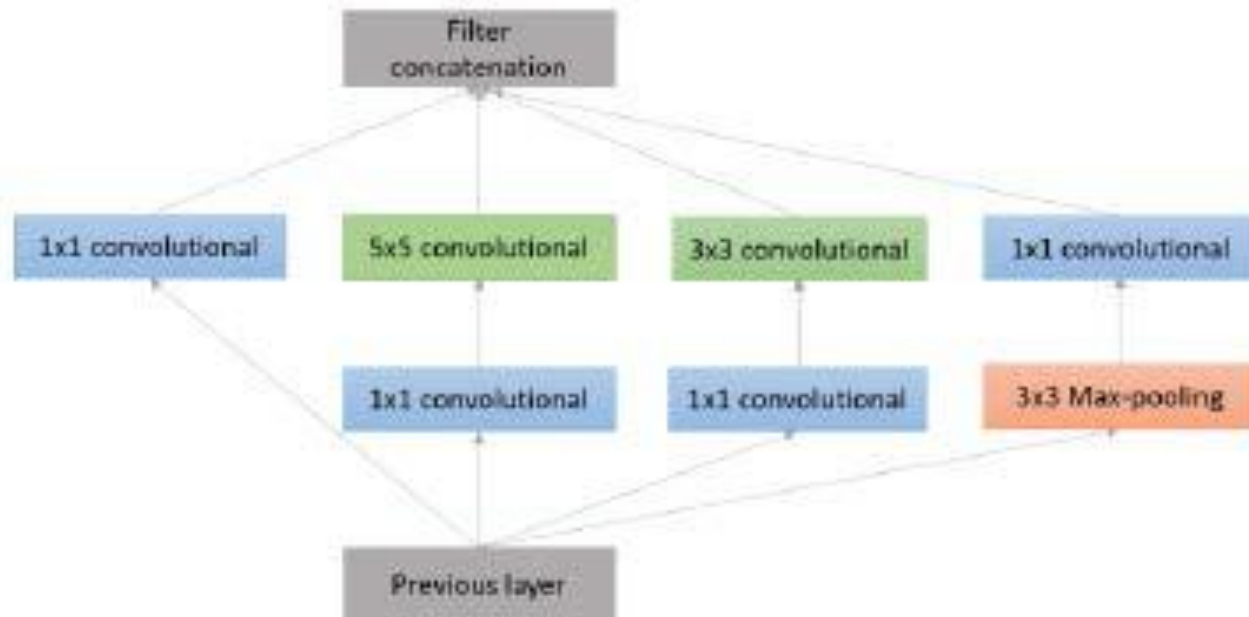
Принципы построения GoogleLeNet

- Изначальный AlexNet делал большие свертки, которые требуют много параметров, в GoogleNet свертки поменьше с большим количеством слоев.
- Уменьшение количества измерений, чтобы компенсировать более толстые слои. Это делается с помощью свертки 1×1 — по сути, линейного фильтра применяющегося по всей картинке, чтобы взять текущее количество измерений, и линейно их смешать в меньшее. Так как эта свертка тоже обучается, получается очень эффективно.
- На каждом уровне применяются несколько convolution kernels разного размера, чтобы вытаскивать признаки разного масштаба. Если масштаб слишком большой для текущего уровня, он распознается на следующем.
- Не делаем hidden FC layers (классификатор) вообще, потому что в них *очень* много параметров. Вместо этого на последнем уровне делаем global average pool и подцепляем его к выходному слою напрямую.

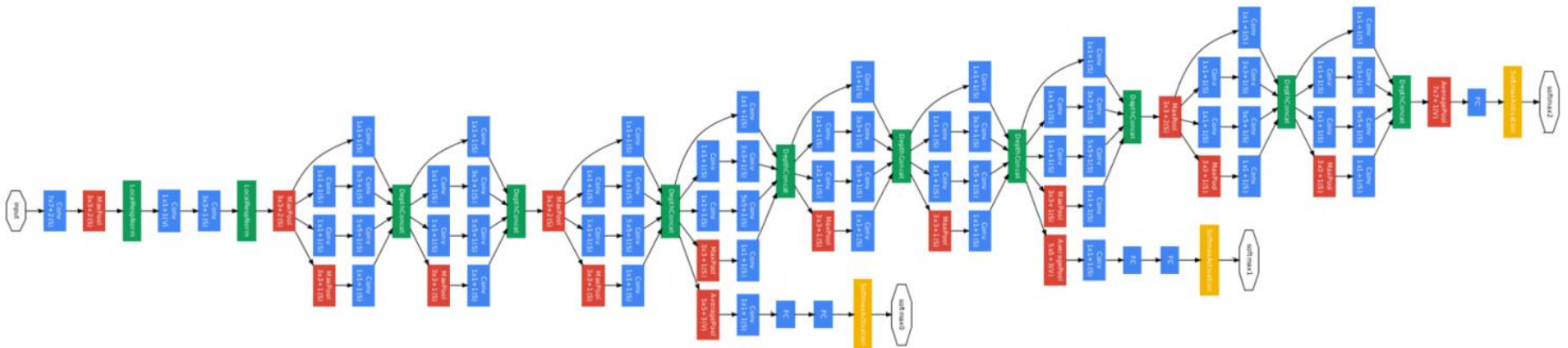
Слой Naïve Inception



Inception final слои



Общая структура GoogleLeNet



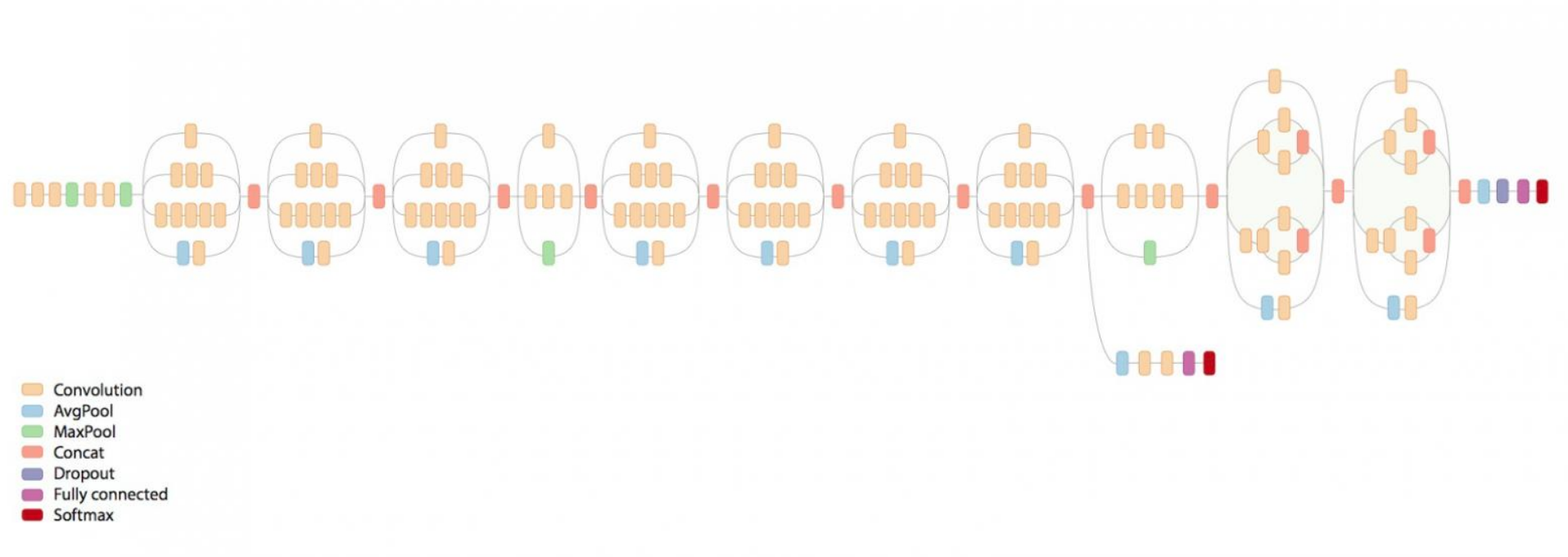
GoogLeNet улучшила современную точность распознавания, используя стек inception слоев.

Разница между naïve inception слоем и final Inception слоем была в добавлении 1x1 сверточного ядра. Эти ядра позволили уменьшить количество уровней, требовательных к вычислениям.

GoogLeNet состояла из 22 слоев, что было намного больше, чем любая другая сеть. Однако количество сетевых параметров GoogLeNet было намного ниже, чем у его предшественника AlexNet или VGG.

GoogLeNet имела 7M параметров сети, когда у AlexNet было 60M и VGG-19 138M. Вычислений для GoogLeNet таким образом было 1.53G MAC намного ниже, чем у AlexNet или VGG.

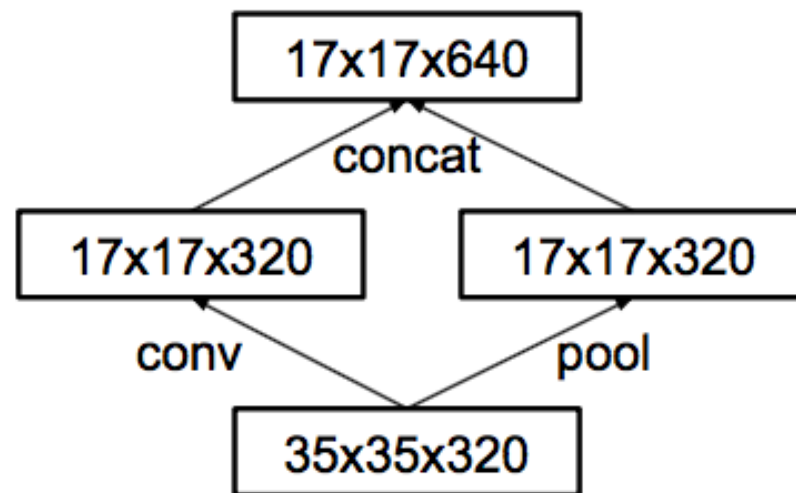
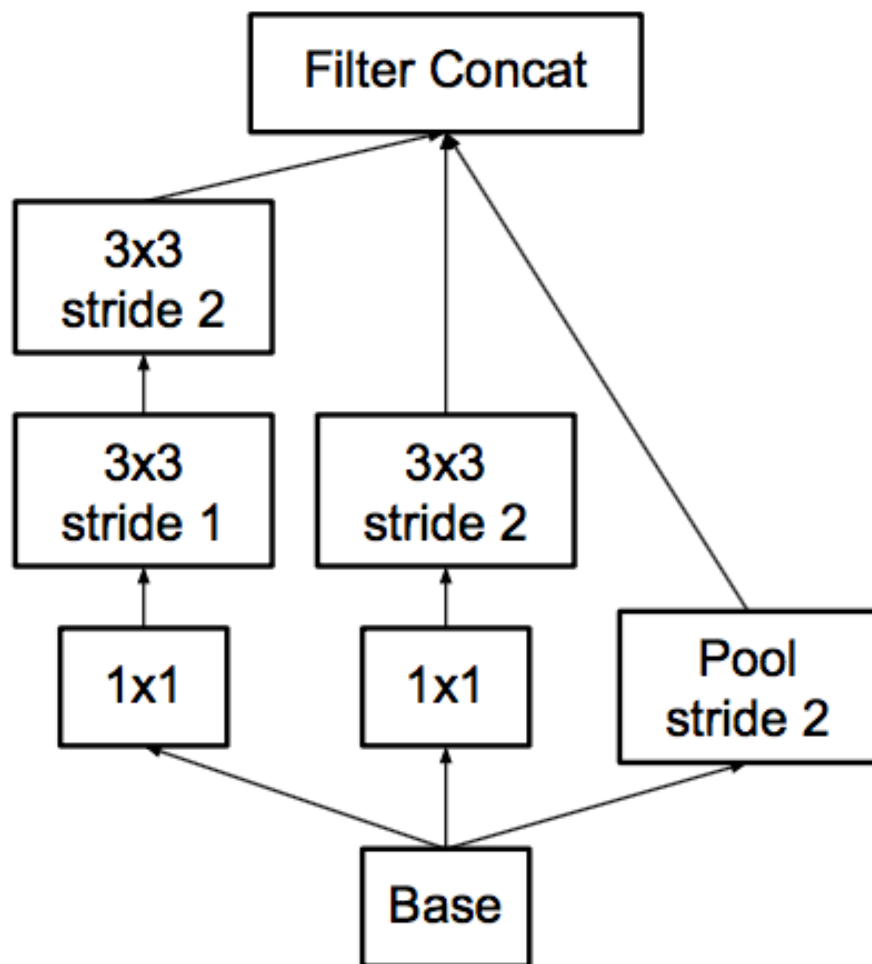
Inception-V3 от Google



Принципы Inception-V3

- Очень много сигналов близки друг к другу в пространстве (т.е. в соседних "пикселах"), и этим можно пользоваться, чтобы делать convolution меньшего размера. Мол, раз соседние сигналы часто скоррелированы, то можно уменьшить размерность перед convolution без потери информации
- Для эффективного использования ресурсов нужно увеличивать и ширину, и глубину сети. Т.е. если ресурсов стало например в два раза больше, эффективнее всего и сделать слои шире, и сеть глубже. Если сделать только глубже, будет неэффективно.
- Плохо иметь резкие bottlenecks, то есть леер с резким уменьшением параметров, особенно в начале
- "Широкие" слои быстрее обучаются, что особенно важно на высоких уровнях (но локально, т.е. вполне можно после них уменьшать размерность)

«Кирпичики» сети Inception-V3



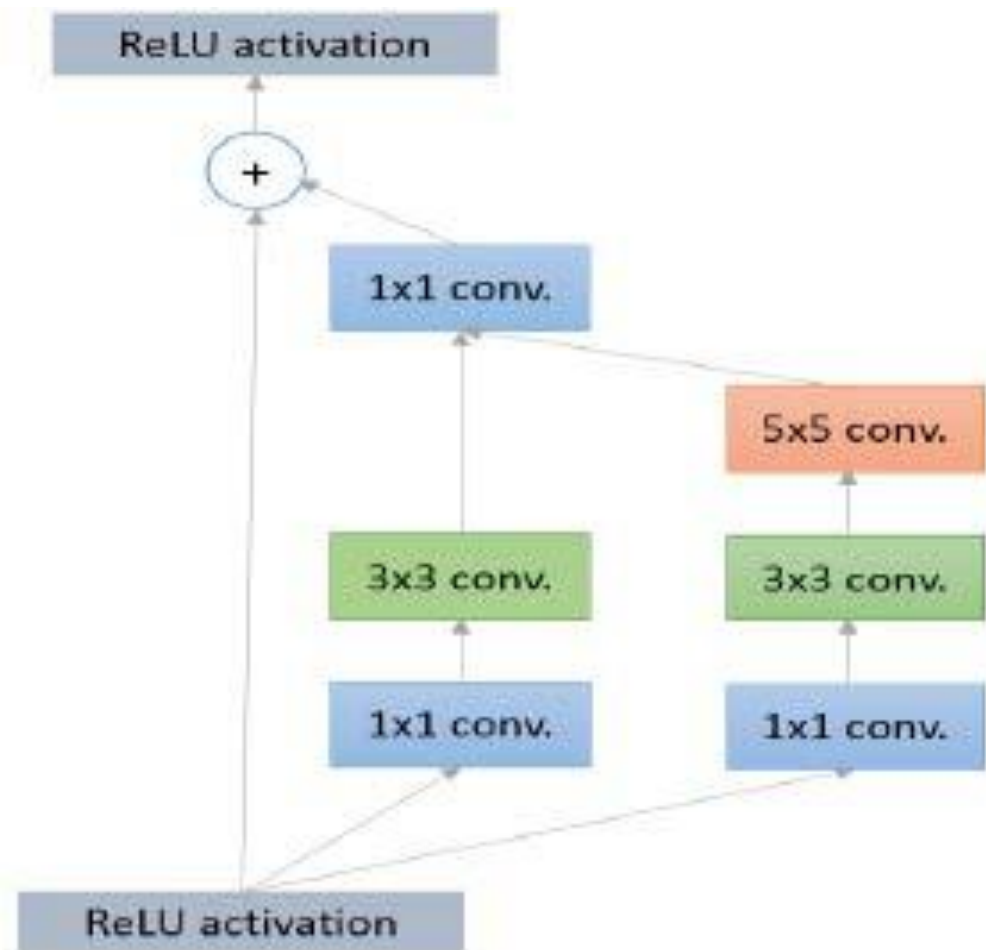
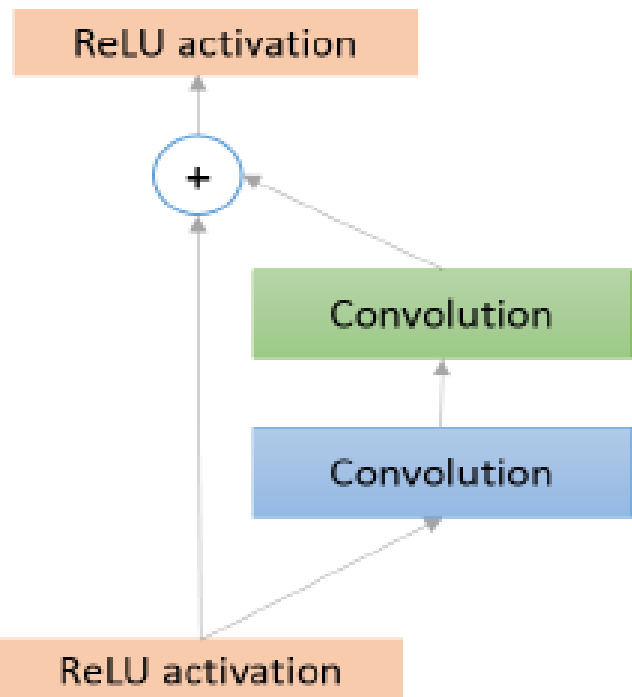
ResNet

Победителем ILSVRC 2015 была архитектура остаточной сети, ResNet (Residual Neural Network). Resnet была разработана Kaiming He с целью разработки ультра-глубоких сетей, которые не страдали бы от проблемы исчезновения градиента, которые имели предшественники.

ResNet разработана с большим количеством слоев; 34, 50, 101, 152 и даже 1202. Популярная ResNet50 содержит 49 слоев свертки и 1 полносвязный уровень в конце сети. Общее количество весов и вычислений MAC для всей сети составляет 25,5 М и 3,9 G соответственно

ResNet представляет собой традиционную сеть прямой передачи с остаточным (residual) подключением.

Базовая схема остаточного блока



Остаточная сеть состоит из нескольких базовых остаточных блоков. Однако операции в остаточном блоке могут варьироваться в зависимости от различной архитектуры остаточных сетей.

Более широкая версия остаточной сети была предложена Загоруйко в 2016 году. Еще один улучшенный метод остаточной сети, известный как агрегированная остаточная трансформация, был предложен в 2016 году.

Недавно были предложены некоторые другие варианты остаточных моделей на основе архитектуры остаточной сети.

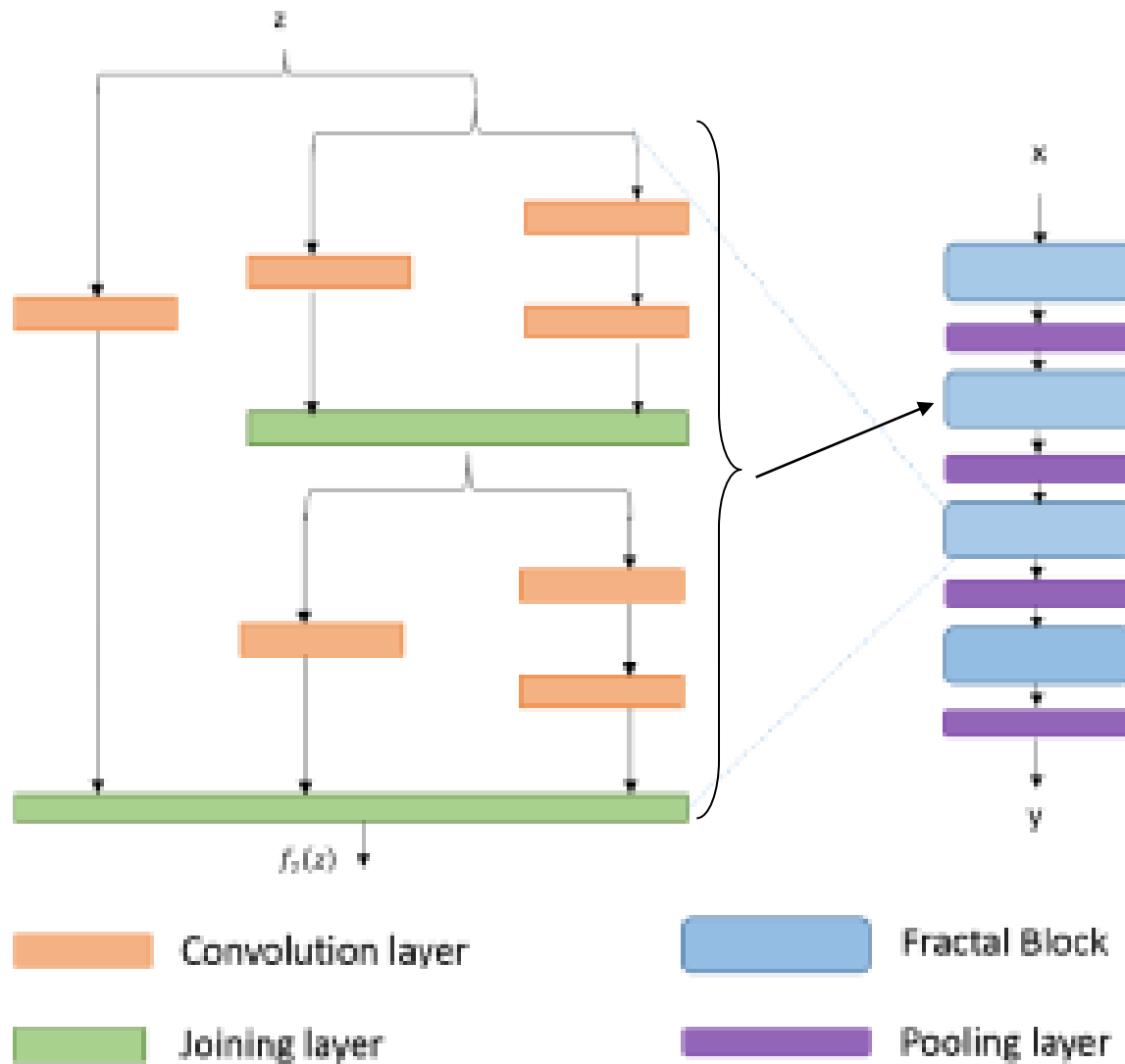
Использование обученной ResNet в Keras

```
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
import numpy as np
model = ResNet50(weights='imagenet')
img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
preds = model.predict(x)
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
# Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), (u'n01871265', u'tusker',
0.1122357), (u'n02504458', u'African_elephant', 0.061040461)]
```

FractalNet

Эта архитектура представляет собой передовую и альтернативную архитектуру модели ResNet, которая эффективна для проектирования больших моделей с номинальной глубиной, но более короткими путями для распространения градиента во время обучения.

Эта концепция основана на обратном пути, который является другим методом регуляризации для создания больших сетей. В результате эта концепция помогает обеспечить компромисс между скоростью и точностью.



Обученные сети в Keras

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
<u>Xception</u>	88	79.0%	94.5%	22.9M	81	109.4	8.1
<u>VGG16</u>	528	71.3%	90.1%	138.4M	16	69.5	4.2
<u>VGG19</u>	549	71.3%	90.0%	143.7M	19	84.8	4.4
<u>ResNet50</u>	98	74.9%	92.1%	25.6M	107	58.2	4.6
<u>ResNet50V2</u>	98	76.0%	93.0%	25.6M	103	45.6	4.4
<u>ResNet101</u>	171	76.4%	92.8%	44.7M	209	89.6	5.2
<u>ResNet101V2</u>	171	77.2%	93.8%	44.7M	205	72.7	5.4
<u>ResNet152</u>	232	76.6%	93.1%	60.4M	311	127.4	6.5
<u>ResNet152V2</u>	232	78.0%	94.2%	60.4M	307	107.5	6.6
<u>InceptionV3</u>	92	77.9%	93.7%	23.9M	189	42.2	6.9
<u>InceptionResNetV2</u>	215	80.3%	95.3%	55.9M	449	130.2	10.0
<u>MobileNet</u>	16	70.4%	89.5%	4.3M	55	22.6	3.4
<u>MobileNetV2</u>	14	71.3%	90.1%	3.5M	105	25.9	3.8

Обученные сети в Keras (2)

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
<u>DenseNet121</u>	33	75.0%	92.3%	8.1M	242	77.1	5.4
<u>DenseNet169</u>	57	76.2%	93.2%	14.3M	338	96.4	6.3
<u>DenseNet201</u>	80	77.3%	93.6%	20.2M	402	127.2	6.7
<u>NASNetMobile</u>	23	74.4%	91.9%	5.3M	389	27.0	6.7
<u>NASNetLarge</u>	343	82.5%	96.0%	88.9M	533	344.5	20.0
<u>EfficientNetB0</u>	29	77.1%	93.3%	5.3M	132	46.0	4.9
<u>EfficientNetB1</u>	31	79.1%	94.4%	7.9M	186	60.2	5.6
<u>EfficientNetB2</u>	36	80.1%	94.9%	9.2M	186	80.8	6.5
<u>EfficientNetB3</u>	48	81.6%	95.7%	12.3M	210	140.0	8.8
<u>EfficientNetB4</u>	75	82.9%	96.4%	19.5M	258	308.3	15.1
<u>EfficientNetB5</u>	118	83.6%	96.7%	30.6M	312	579.2	25.3
<u>EfficientNetB6</u>	166	84.0%	96.8%	43.3M	360	958.1	40.4
<u>EfficientNetB7</u>	256	84.3%	97.0%	66.7M	438	1578.9	61.6

Обученные сети в Keras (3)

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-	-
EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-	-
EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-	-
EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-	-
EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-	-
EfficientNetV2M	220	85.3%	97.4%	54.4M	-	-	-
EfficientNetV2L	479	85.7%	97.5%	119.0M	-	-	-
ConvNeXtTiny	109.42	81.3%	-	28.6M	-	-	-
ConvNeXtSmall	192.29	82.3%	-	50.2M	-	-	-
ConvNeXtBase	338.58	85.3%	-	88.5M	-	-	-
ConvNeXtLarge	755.07	86.3%	-	197.7M	-	-	-
ConvNeXtXLarge	1310	86.7%	-	350.1M	-	-	-

Генеративные состязательные сети (GAN)

Понятие генеративных состязательных сетей (Generative Adversarial Networks) было введено в 2014 (Goodfellow). Этот тип сети используется для генерации изображений, которые сама сеть не может отличить от настоящих.

Применение данных сетей широко исследовалось для генерации различных изображений цифр, тематических изображений и лиц.

Изначально сеть-генератор получает на вход некоторый случайный вектор или матрицу небольшого размера и, постепенно применяя операцию *развертки*, возвращает сгенерированное изображение нужного размера. Операция развертки, которую также начали называть операцией деконволюции, является операцией, применяющей ко входному слою градиент от обычной свертки.

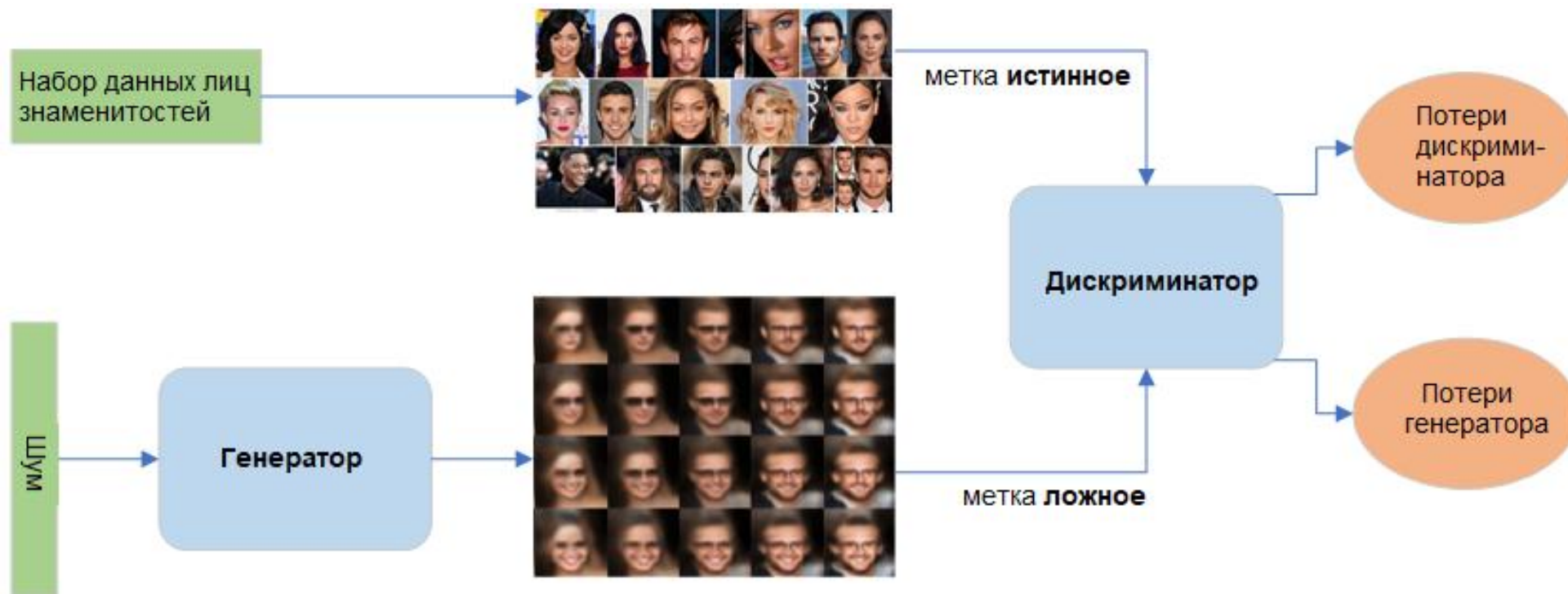
Сеть-дискриминатор принимает на вход изображение и пытается определить, взято ли это изображение из обучающей выборки, или это изображение, полученное из генератора. Таким образом, дискриминатор является обычной сверточной нейронной сетью, классифицирующей изображения на два класса: real и fake.

Генеративные состязательные сети (2).

Пример – генерация лиц знаменитостей.

- Генеративная часть отвечает за ввод N -мерных равномерно распределенных случайных чисел (шума) и генерацию ложных лиц. Генератор усваивает вероятность $P(X)$, где X – это входные данные.
- Дискриминационная часть – это простой классификатор, оценивающий и отделяющий сгенерированные лица от истинных лиц знаменитостей. Дискриминатор усваивает условную вероятность $P(Y | X)$, где X – это входные данные, а Y – метка.
- Генеративная сеть обучается с целью максимизации итоговой ошибки классификации (между истинными и сгенерированными данными), а дискриминационная сеть обучается с целью минимизации этой ошибки. Вот откуда происходит термин "состязательная сеть".

Генеративные состязательные сети (3). Пример – генерация лиц знаменитостей



<https://proglib.io/p/generativnaya-sostyazatel'naya-set-gan-dlya-chaynikov-poshagovoe-rukovodstvo-2021-07-16>

Генеративные состязательные сети (4)

- При обучении дискриминатора мы игнорируем потери генератора и используем только функцию потерь дискриминатора, которая штрафует дискриминатор за неверную классификацию сгенерированных лиц как истинных или истинных лиц как фейковых. Веса дискриминатора обновляются методом обратного распространения (backpropagation). Веса генератора не обновляются.
- При обучении генератора мы используем функцию потерь генератора, штрафующую генератор за неудачный обман дискриминатора, то есть за генерацию лица, которое дискриминатор признал фейковым. В процессе обучения генератора дискриминатор заморожен, и только веса генератора обновляются методом обратного распространения.

Дискриминатор в Keras

```
from keras.layers import Conv2D, BatchNormalization, Input, GlobalAveragePooling2D, Dense
from keras.models import Model
from keras.layers.advanced_activations import LeakyReLU
# функция для создания слоев дискриминатора
def build_discriminator(start_filters, spatial_dim, filter_size):
# функция создания блока CNN block для уменьшения размера изображения
def add_discriminator_block(x, filters, filter_size):
    x = Conv2D(filters, filter_size, padding='same')(x)
    x = BatchNormalization()(x)
    x = Conv2D(filters, filter_size, padding='same', strides=2)(x)
    x = BatchNormalization()(x)
    x = LeakyReLU(0.3)(x)
    return x
# ввод - это изображение с размерами spatial_dim x spatial_dim и 3 каналами
inp = Input(shape=(spatial_dim, spatial_dim, 3))
# строим дискриминатор для уменьшения изображения в 4 раза
x = add_discriminator_block(inp, start_filters, filter_size)
x = add_discriminator_block(x, start_filters * 2, filter_size)
x = add_discriminator_block(x, start_filters * 4, filter_size)
x = add_discriminator_block(x, start_filters * 8, filter_size)
# усреднение и возврат бинарного вывода x = GlobalAveragePooling2D()(x)
x = Dense(1, activation='sigmoid')(x)
return Model(inputs=inp, outputs=x)
```

Генератор в Keras

```
from keras.layers import Deconvolution2D, Reshape
def build_generator(start_filters, filter_size, latent_dim):
# функция для создания блока CNN, увеличивающего размеры изображения
def add_generator_block(x, filters, filter_size):
x = Deconvolution2D(filters, filter_size, strides=2, padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU(0.3)(x)
return x
# вход - это вектор шума
inp = Input(shape=(latent_dim,))
# проекция вектора шума в тензор с такой же размерностью,
# как последний сверточный слой дискриминатора
x = Dense(4 * 4 * (start_filters * 8), input_dim=latent_dim)(inp)
x = BatchNormalization()(x)
x = Reshape(target_shape=(4, 4, start_filters * 8))(x)
# строим генератор для увеличения изображения в 4 раза
x = add_generator_block(x, start_filters * 4, filter_size)
x = add_generator_block(x, start_filters * 2, filter_size)
x = add_generator_block(x, start_filters, filter_size)
x = add_generator_block(x, start_filters, filter_size)
# превращаем вывод в трехмерный тензор, изображение с 3 каналами
x = Conv2D(3, kernel_size=5, padding='same', activation='tanh')(x)
return Model(inputs=inp, outputs=x)
```

Полная модель GAN

```
import pandas as pd
import os
from keras.optimizers import Adam
# загружаем атрибуты изображений знаменитостей
df_celeb = pd.read_csv('list_attr_celeba.csv')
TOTAL_SAMPLES = df_celeb.shape[0]
# мы будем уменьшать изображения
SPATIAL_DIM = 64
# размер вектора шума
LATENT_DIM_GAN = 100
# размер фильтра в сверточных слоях
FILTER_SIZE = 5
# количество фильтров в сверточном слое
NET_CAPACITY = 16
# размер пакета
BATCH_SIZE_GAN = 32
# интервал для отображения сгенерированных изображений
PROGRESS_INTERVAL = 80
# директория для хранения сгенерированных изображений
ROOT_DIR = 'visualization' if not os.path.isdir(ROOT_DIR):
os.mkdir(ROOT_DIR)
def construct_models(verbose=False):
#### дискриминатор
discriminator = build_discriminator(NET_CAPACITY, SPATIAL_DIM, FILTER_SIZE)
# компилируем дискриминатор
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002), metrics=['mae'])
#### генератор # не компилируем генератор
generator = build_generator(NET_CAPACITY, FILTER_SIZE, LATENT_DIM_GAN)
#### DCGAN
gan = Sequential()
gan.add(generator)
gan.add(discriminator)
discriminator.trainable = False
gan.compile(loss='binary_crossentropy', optimizer=Adam(lr=0.0002), metrics=['mae'])
if verbose: generator.summary()
discriminator.summary()
gan.summary()
return generator, discriminator, gan
generator_celeb, discriminator_celeb, gan_celeb = construct_models(verbose=True)
```

Схема обучения GAN

- Шаг 1. Выбираем несколько реальных изображений из тренировочного набора
- Шаг 2. Генерируем несколько фейковых изображений. Для этого мы создаем несколько случайных векторов шума и создаем из них изображения с помощью генератора.
- Шаг 3. Обучаем дискриминатор на протяжении одной или большего количества эпох, используя как реальные, так и фейковые изображения. При этом будут обновляться только веса дискриминатора, поскольку мы пометим все реальные изображения как 1, а фейковые как 0.
- Шаг 4.** Создаем еще несколько фейковых изображений.
- Шаг 5.** Обучаем полную модель GAN на протяжении одной или большего количества эпох, используя только фейковые изображения. При этом будут обновляться только веса генератора, а всем фейковым изображениям будет назначена метка 1.

Сгенерированные лица



Для сравнения полученные при
Работе автоэнкодера ->



Возможно, наиболее убедительное применение GAN – это GAN для задач, которые требуют генерации новых примеров. Здесь, Ян Гудфеллоу указывает три главных задачи:

- Создание изображения с супер-разрешением. Возможность генерировать версии входных изображений с высоким разрешением.
- Создание произведений искусства. Способность к созданию великолепных новых художественных изображений, эскизов, живописи и многого другого.
- Перевод изображения в изображение. Возможность переводить фотографии между доменами, такими как день и ночь, лето и зима и многие другие.