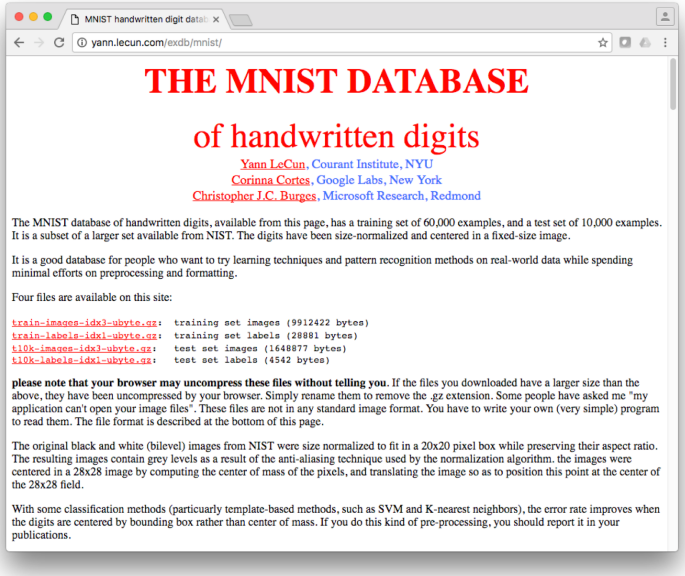
Introduction to Convolutional Neural Networks

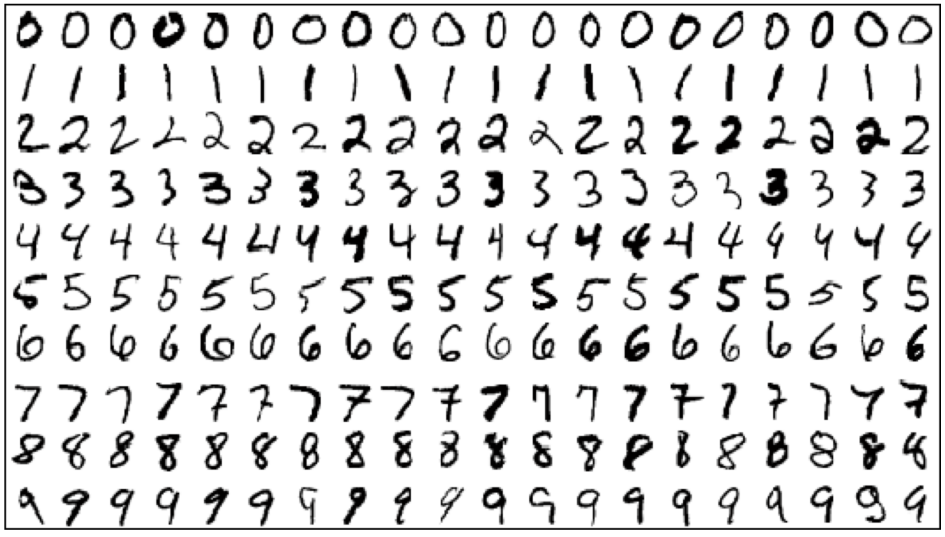
Wen Phan

June 8, 2017

Outline

MNIST Database





MNIST Digit

- 28 x 28 pixels

```
Shape: (28, 28)
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0  26 121 134 253 253 255 127  13   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0 121 242 252 252 252 252 253 252  82   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0  90 142 226 247 252 252 252 252 252 253 252 242  85   0   0   0   0   0   0   0   0   0   0
  0   0   0   0 102 250 252 252 252 235 172  88  39  39 211 252 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0 121 252 252 231 150  42   0   0   0   0  89 249 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0  95 198 149  45   0   0   0   0   0   0   0 240 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 240 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 240 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0  70 247 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 121 252 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 121 253 253 173   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 121 252 252 172   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 240 252 252  53   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0  43 253 252 252 107  81 109 192  81  34   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0  46 227 253 252 252 252 252 252 252 252 140   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   6 169 252 253 252 252 252 252 252 252 252 248  89   0   0   0
  0   0   0   0   0   0   0   0   0   0   0  48 252 252 253 252 252 218 107  79 176 252 250 100   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0 160 252 252 253 252 178  15   0   0  36 225 143   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0 160 252 252 253 183  11   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0 160 252 252 190   6   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

- Image as a pixel vector or 784 features: $\mathbf{x}^T = [x_1, \ldots, x_{784}]$

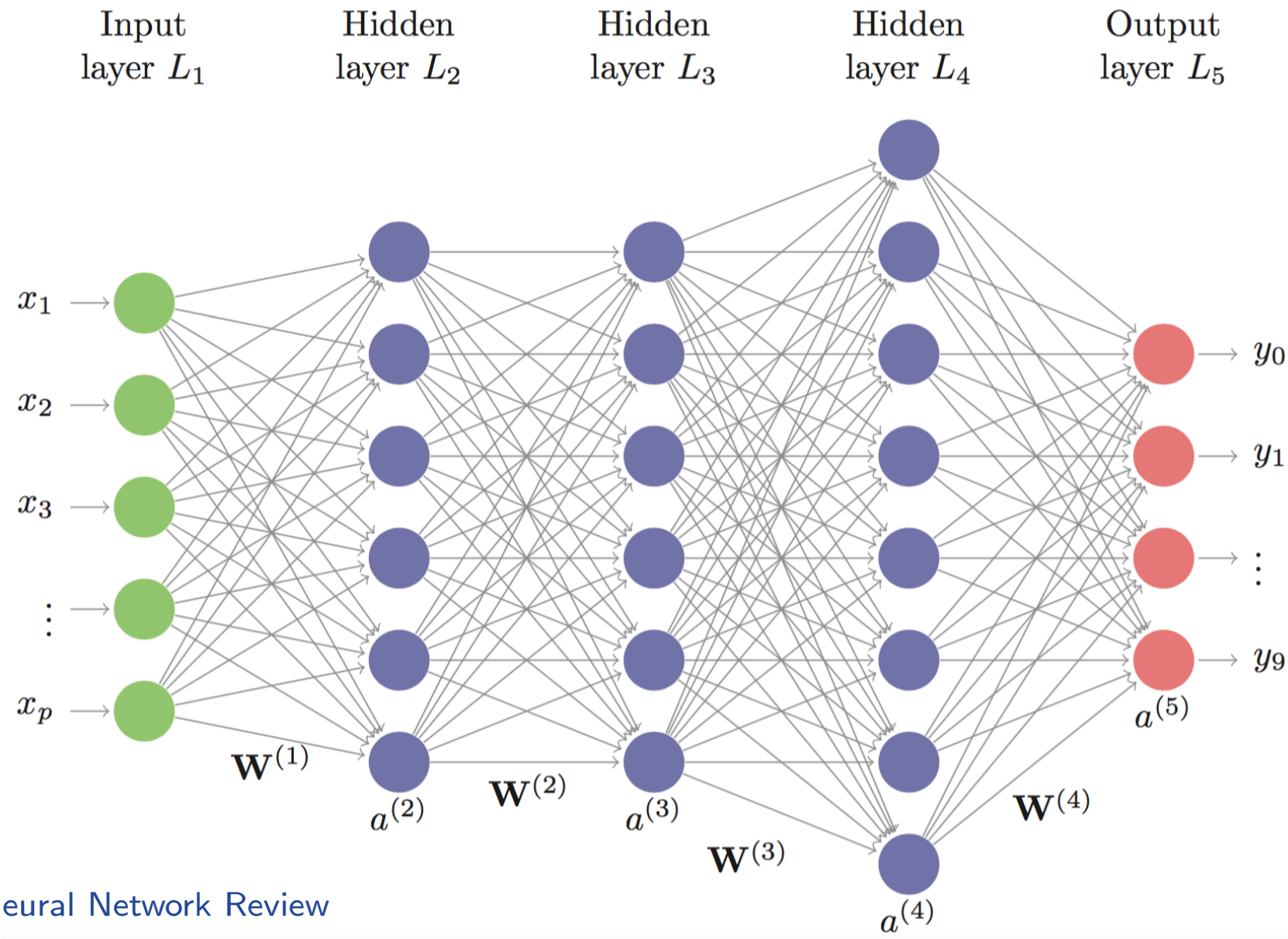
MNIST Data Set

$$\mathbf{X} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,P} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \cdots & x_{N,P} \end{bmatrix} = \begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix}$$

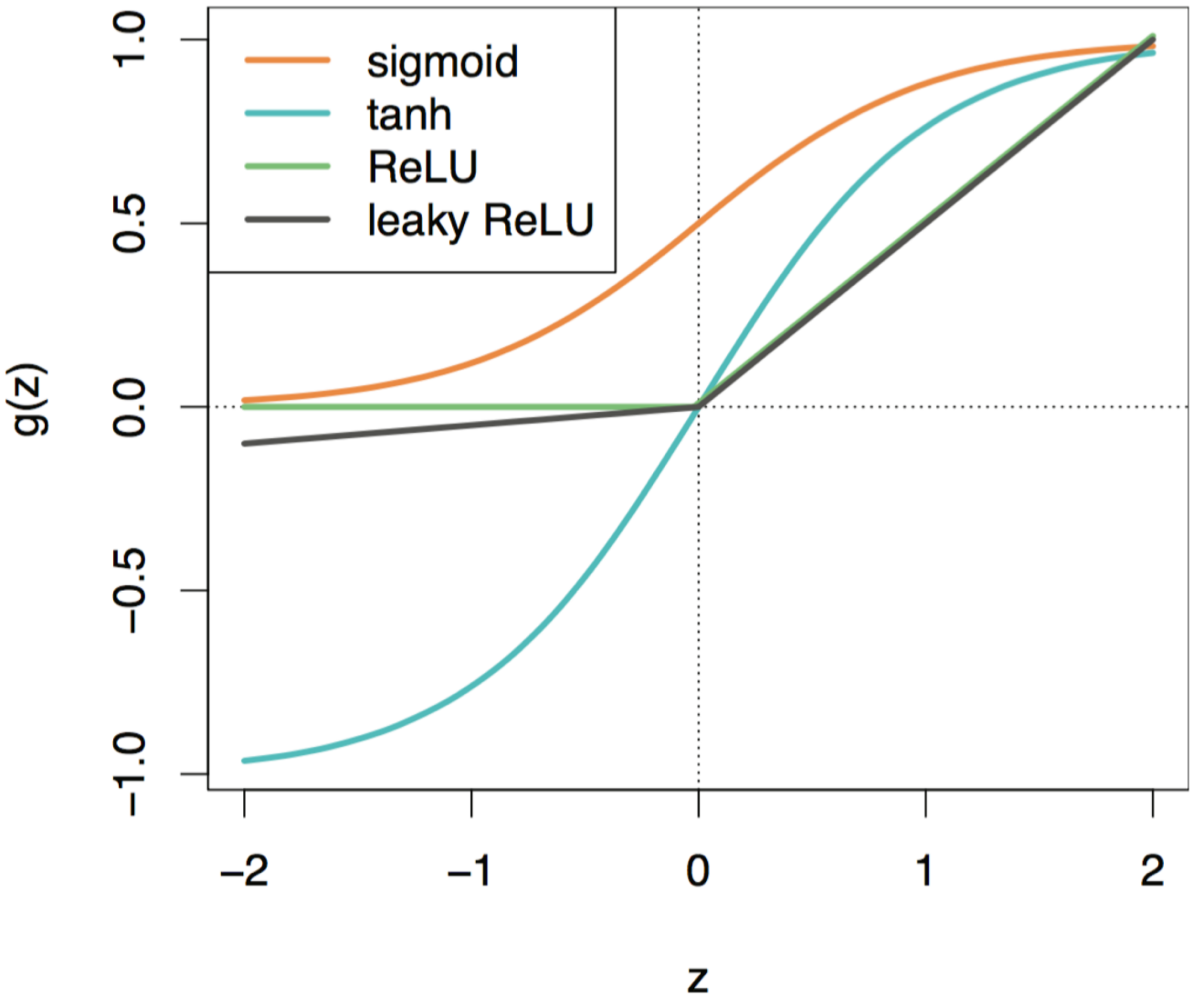
, where $\mathbf{x}_i^T = [x_{i,1}, \ldots, x_{i,P}]$ is the i-th data record

- Training set: $N = 60,000$
- Validation/test set: $N = 60,000$
- Output classes: 10

MLP Feedforward Network



Activations



Activations

Training gradient descent with saturating nonlinearities is slower than non-saturating nonlinearities

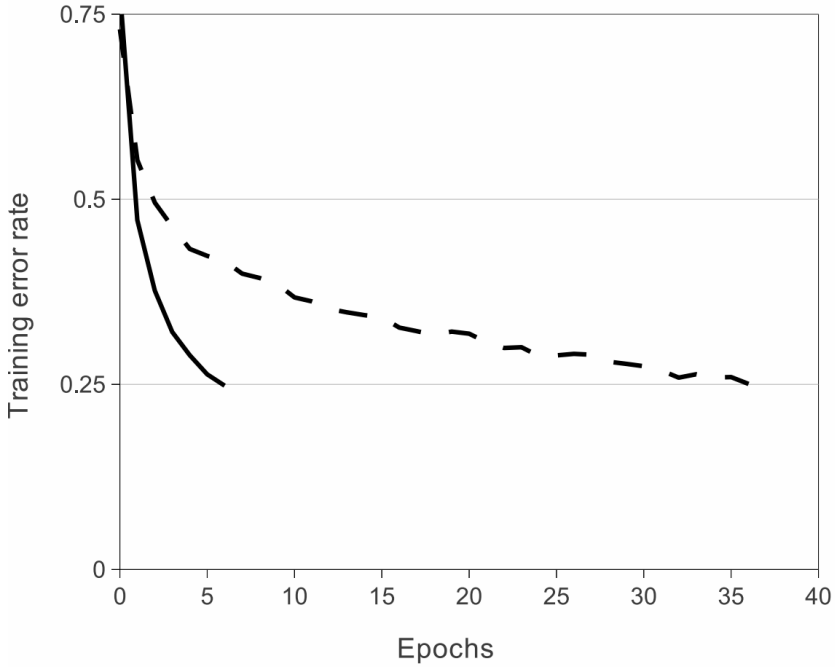


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

Output Layer Requirements

- All probabilities must sum up to one.
  $$\sum_{k \in K} \mathbf{Pr}\left(y = k | \mathbf{x}\right) = 1$$
- Probabilities must be between 0 and 1.
  $$0 \leq \mathbf{Pr}\left(y = k | \mathbf{x}\right) \leq 1, k \in K$$

Softmax Output Layer

$$g_k(\cdot) = \mathbf{softmax}_k(a_1, a_2, \ldots, a_{|K|})$$

$$\mathbf{softmax}_k(a_1, a_2, \ldots, a_{|K|}) = \frac{\exp(a_k)}{\sum_{j \in K} \exp(a_j)}$$

$$= \frac{\text{evidence for class } k}{\text{total evidence for all classes}}$$

Optimization

$$\underset{\mathbf{W},\mathbf{b}}{\text{minimize}} \; -\frac{1}{N}\sum_{i=1}^{N}\left[\sum_{j=1}^{K} y_j \log \hat{y}_j\right]$$

- Optimization Algorithm: Gradient Descent
  - Learning Rate (α): 0.5

Gradient

- Slope or rate of change of function
- Direction of greatest change

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_p} \end{bmatrix}, \mathbf{x} \in \mathbb{R}^p$$

- Example:

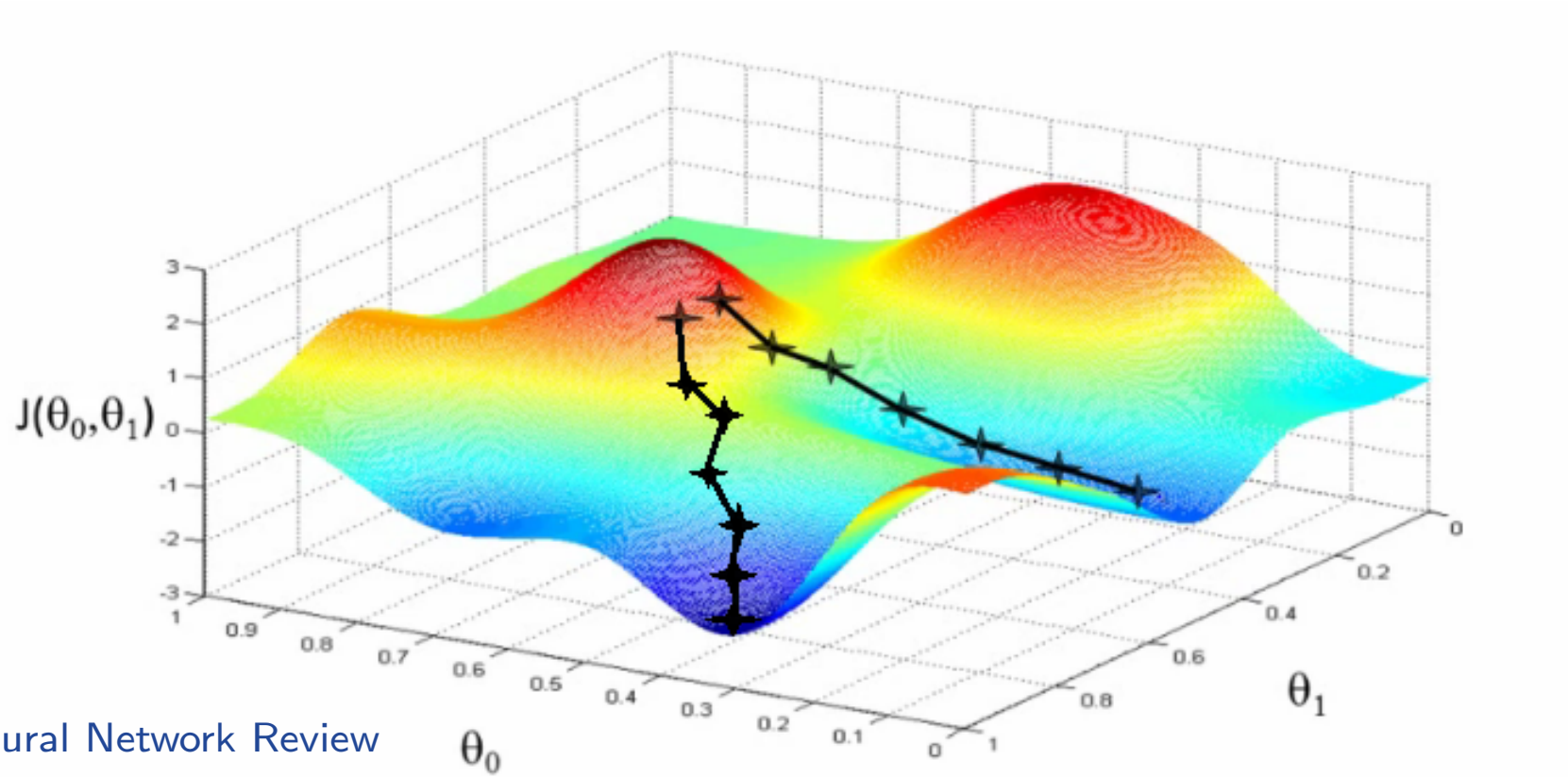
$$f(x_1, x_2) = x_1^2 + x_2^2$$

$$\nabla f(x_1, x_2) = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}$$

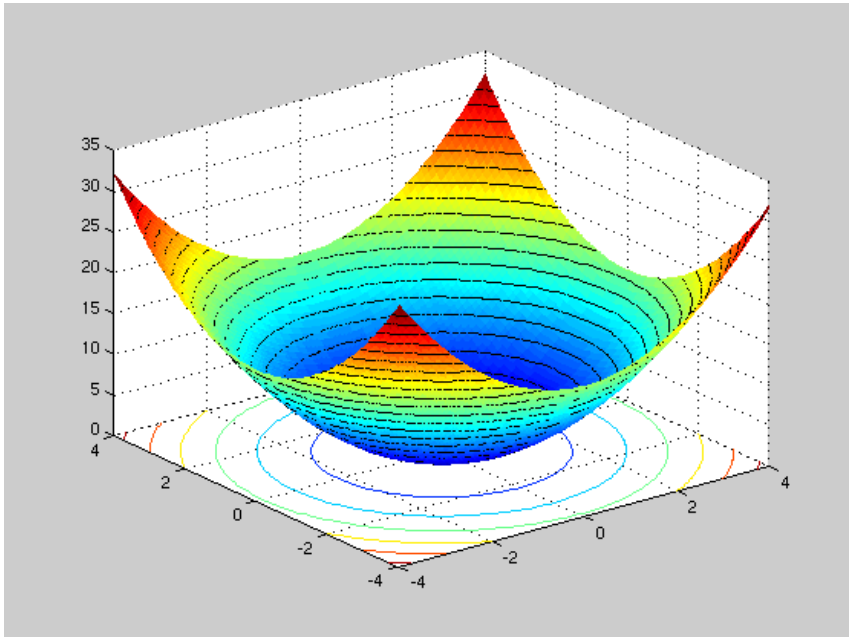
Gradient Descent

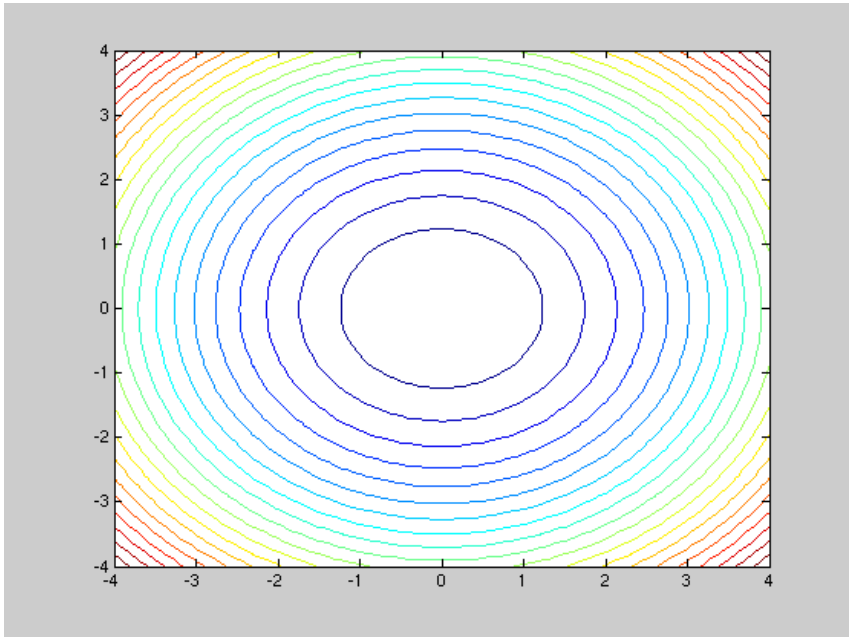
- Gradient provides direction to update parameters (θ).
- Learning rate (α): Update "step size".

1: **repeat**
2: $\quad \theta' := \theta - \alpha \nabla J(\theta)$
3: **until** convergence

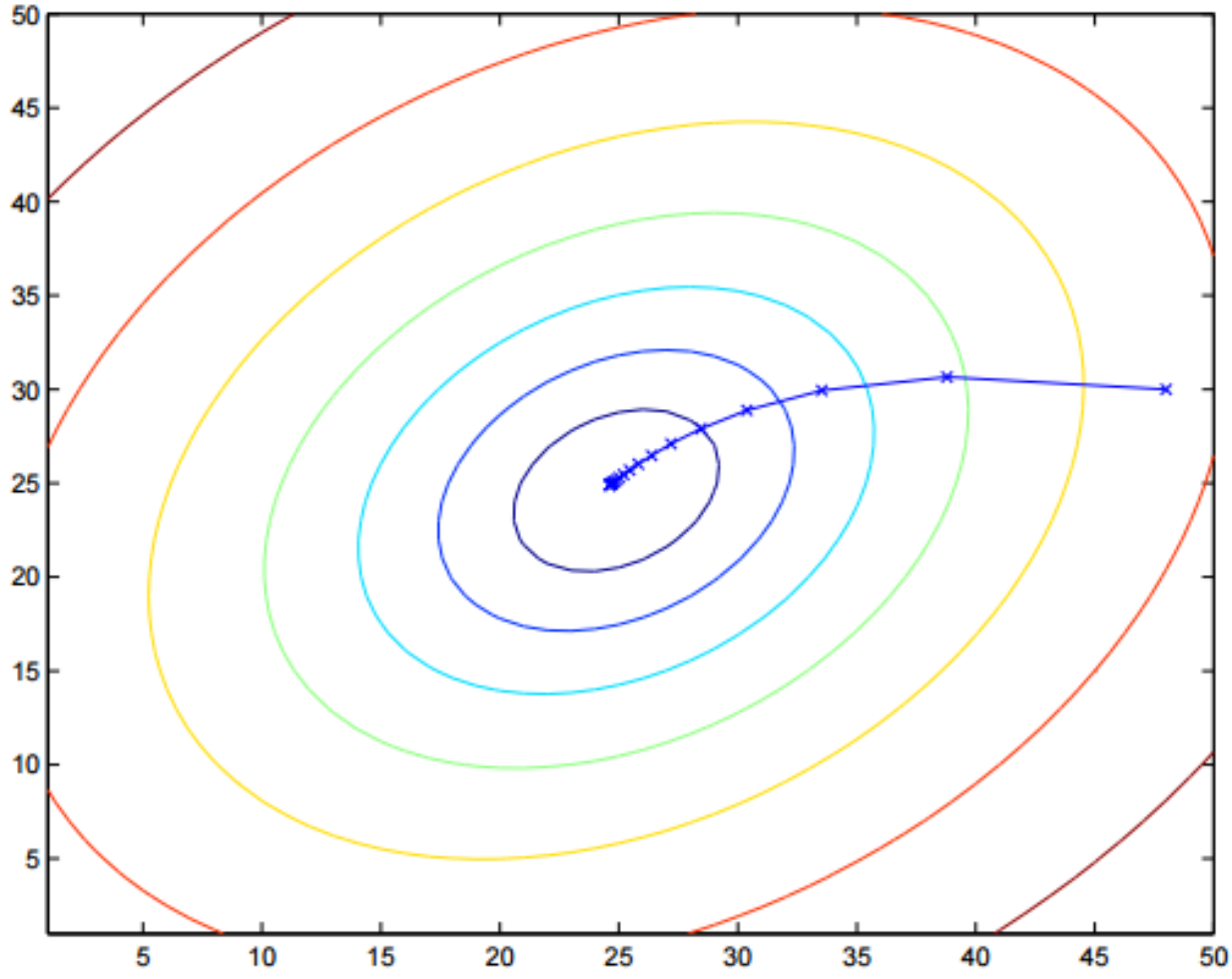


Paraboloid Surface





Gradient Descent via Contour Plot



Outline

Keras

- https://github.com/fchollet/keras

Sequential Model

Typical pattern is:

- `model = Sequential()`
- `model.add(...)`
- `model.compile(...)`
- `model.fit(...)`
- `model.evaluate(...)`

Sequential API

- `compile(self, optimizer, loss, metrics=None, sample_weight_mode=None)`
- `fit(self, x, y, batch_size=32, epochs=10, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0)`
- `evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)`
- `predict(self, x, batch_size=32, verbose=0)`
- `layers`
- `get_layer(self, name=None, index=None)`

Input Specification

- `input_shape`: Argument to layer. Tuple. Batch dimension is not included.
- `input_dim` and `input_length`
- `batch_size`

Compilation

- **Optimizer**: String identifier of existing optimizer or instance of `Optimzer` class.
- **Loss Function**: String identifier of existing loss function or objective function.
- **Metrics**: List of metrics. String identifier of existing metric or custom metric function.

Compilation Examples

```
# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For custom metrics
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Training

```
fit(
        self,
        x, y,
        batch_size=32,
        epochs=10,
        verbose=1,
        callbacks=None,
        validation_split=0.0,
        validation_data=None,
        shuffle=True,
        class_weight=None,
        sample_weight=None,

        initial_epoch=0)
```

Evaluate

```
evaluate(
        self,
        x, y,
        batch_size=32,
        verbose=1,
        sample_weight=None)
```

Layer Class

- `Layer` class (abstract base layer class): `https://github.com/fchollet/keras/blob/master/keras/engine/topology.py`

Layer Class Properties

- name: String, must be unique within a model.
- input_spec: List of InputSpec class instances
- trainable: Boolean, whether the layer weights will be updated during training.
- uses_learning_phase: Whether any operation of the layer uses ‘K.in_training_phase()‘ or ‘K.in_test_phase()‘.
- input_shape: Shape tuple.
- output_shape: Shape tuple. See above.
- inbound_nodes: List of nodes.
- outbound_nodes: List of nodes.
- input, output: Input/output tensor(s).
- input_mask, output_mask: Same as above, for masks.
- trainable_weights: List of variables.
- non_trainable_weights: List of variables.
- weights: The concatenation of the lists trainable_weights and non_trainable_weights (in this order).
- constraints: Dict mapping weights to constraints.

Layer Class Methods

- call(x, mask=None): Where the layer's logic lives.
- get_weights()
- set_weights(weights)
- get_config()
- count_params()
- compute_output_shape(input_shape)
- compute_mask(x, mask)
- get_input_at(node_index)
- get_output_at(node_index)
- get_input_shape_at(node_index)
- get_output_shape_at(node_index)
- get_input_mask_at(node_index)
- get_output_mask_at(node_index)

Common Layer Methods

- `layer.get_weights()`: returns the weights of the layer as a list of Numpy arrays.
- `layer.set_weights(weights)`: sets the weights of the layer from a list of Numpy arrays (with the same shapes as the output of `get_weights`).
- `layer.get_config()`: returns a dictionary containing the configuration of the layer.
- **Tensor Info**: `layer.input`, `layer.output`, `layer.input_shape`, `layer.output_shape`
- **Tensor Info for Shared Layer**:
  `layer.get_input_at(node_index)`,
  `layer.get_output_at(node_index)`,
  `layer.get_input_shape_at(node_index)`,
  `layer.get_output_shape_at(node_index)`

Keras Core Layers

- `Dense`
- `Activation`
- `Dropout`
- `Flatten`

Dense Layer

```
keras.layers.core.Dense(
        units,
        activation=None,
        use_bias=True,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        kernel_regularizer=None,
        bias_regularizer=None,
        activity_regularizer=None,
        kernel_constraint=None,
        bias_constraint=None)
```

Actvation Layer

`keras.layers.core.Activation(activation)`

- `softmax`
- `softsign`
- `relu`
- `tanh`
- `sigmoid`
- `hard_sigmoid`
- `linear`

Dropout Layer

```
keras.layers.core.Dropout(rate,
        noise_shape=None,
        seed=None)
```

Flatten Layer

```
keras.layers.core.Flatten()

model = Sequential()
model.add(Convolution2D(64, 3, 3,
        border_mode='same',
        input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

Keras Metrics

- `binary_accuracy`
- `categorical_accuracy`
- `sparse_categorical_accuracy`
- `top_k_categorical_accuracy`

Other Layers

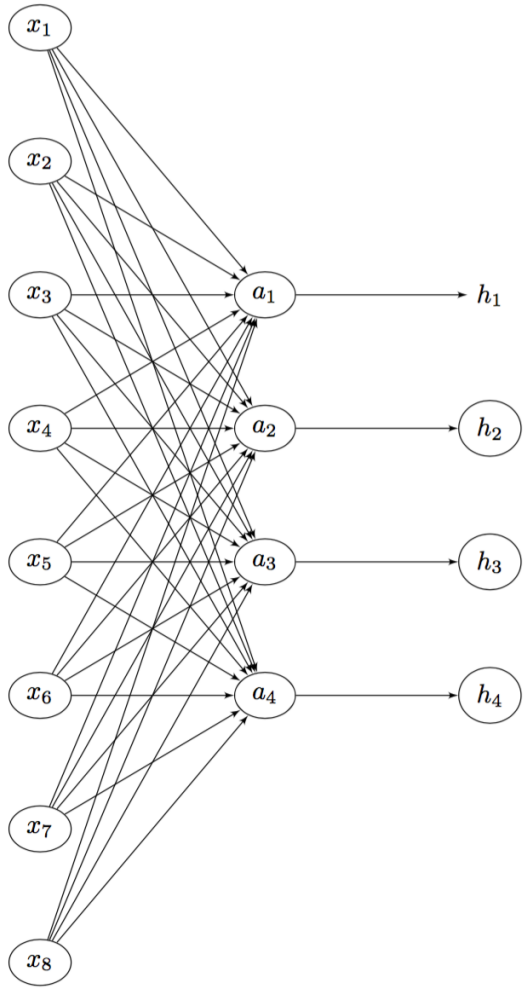
- **Losses**: `mean_squared_error` `mean_absolute_error`, `mean_absolute_percentage_error`, `mean_squared_logarithmic_error`, `squared_hinge`, `hinge`, `logcosh`, `categorical_crossentropy`, `sparse_categorical_crossentropy`, `binary_crossentropy`, `kullback_leibler_divergence`, `poisson`, `cosine_proximity`
- **Optimizers**: `SGD`, `RMSprop`, `Adagrad`, `Adadelta`, `Adam`, `Adamax`, `Nadam`

Other Layers (cont.)

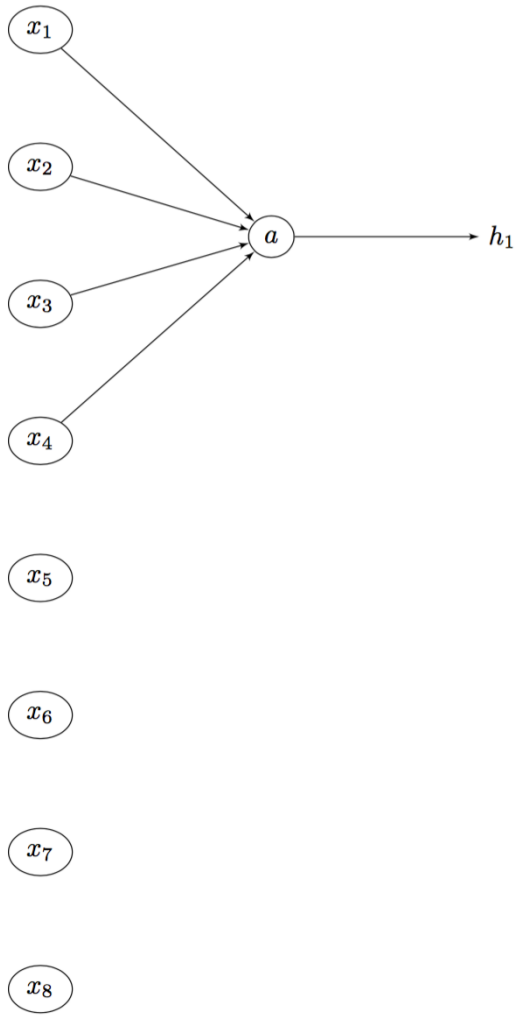
- **Regularization**
  - **Regularizers**: `kernel_regularizer`, `bias_regularizer`, `activity_regularizer`
  - **Penalties**: L1 (`keras.regularizers.l1(0.)`), L2 (`keras.regularizers.l2(0.)`), Elastic Net (`keras.regularizers.l1_l2(0.)`)
- **Constraints**
  - **Parameters**: `kernel_constraint`, `bias_constraint`
  - **Types**: `max_norm(max_value=2, axis=0)`, `non_neg()`, `unit_norm()`

Outline

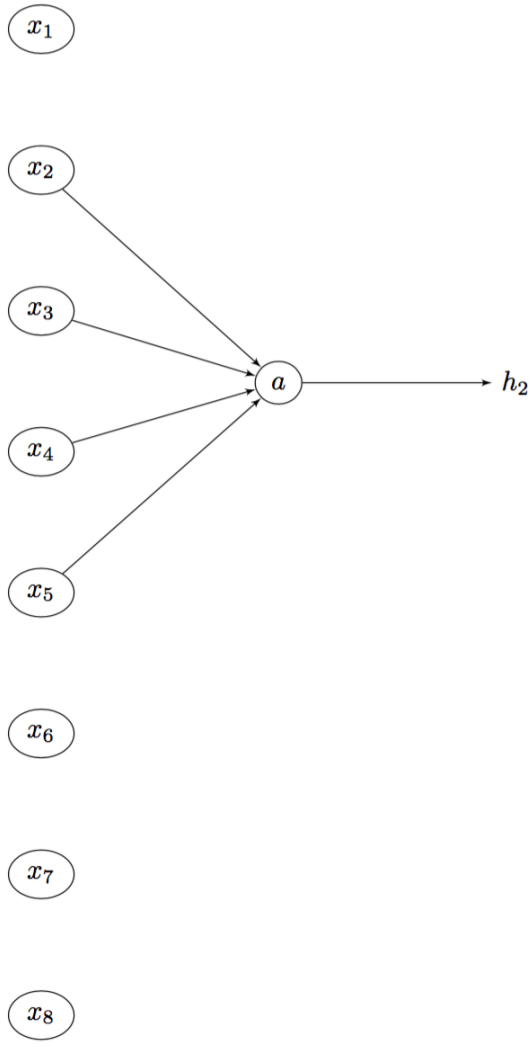
Fully-Connected Layer



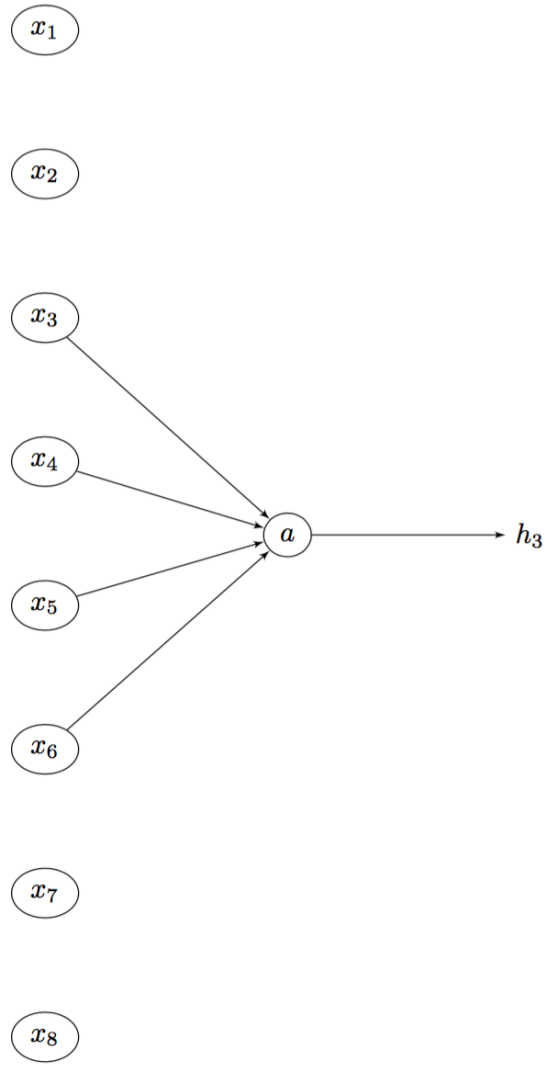
One-Dimensional Convolution



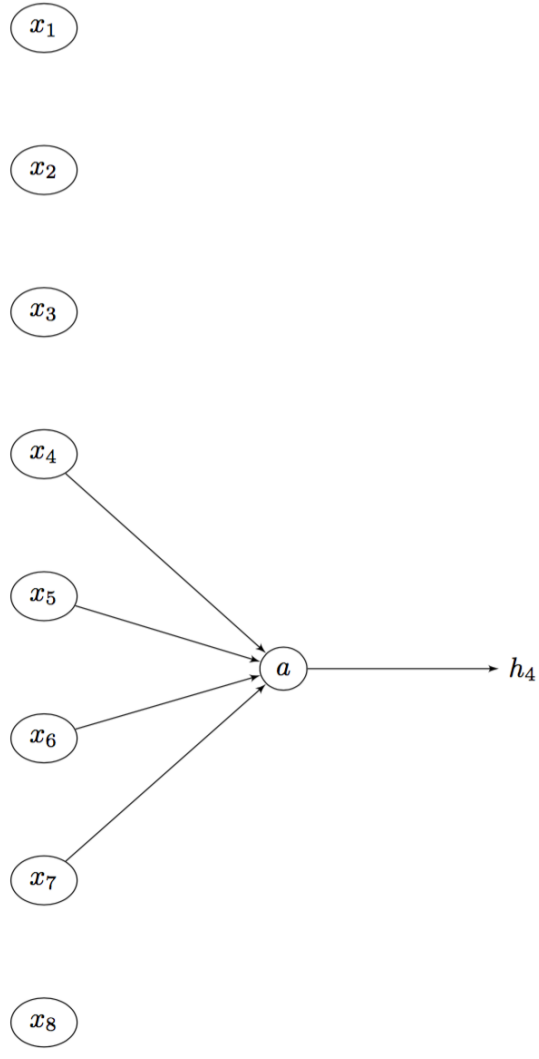
One-Dimensional Convolution



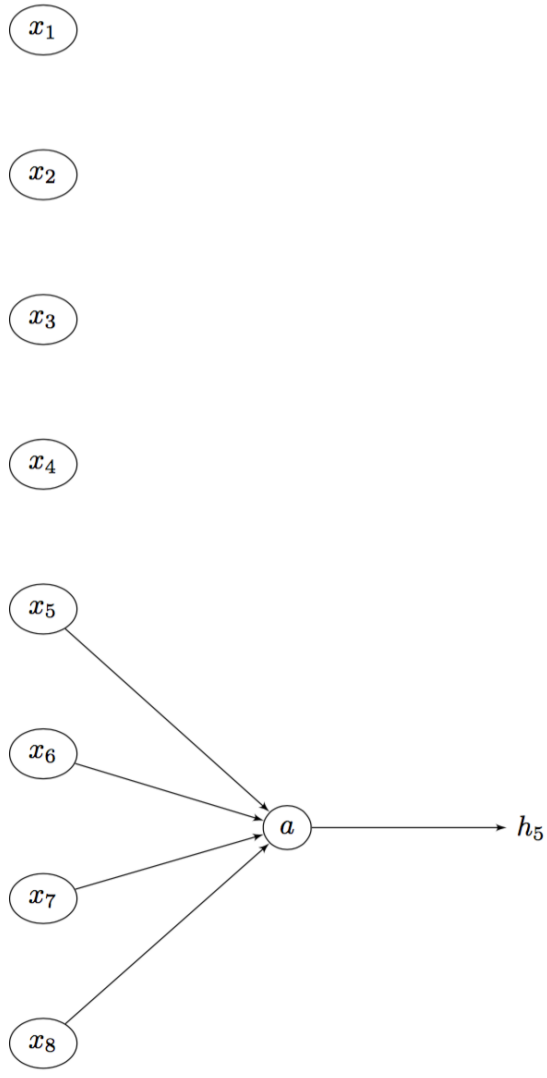
One-Dimensional Convolution



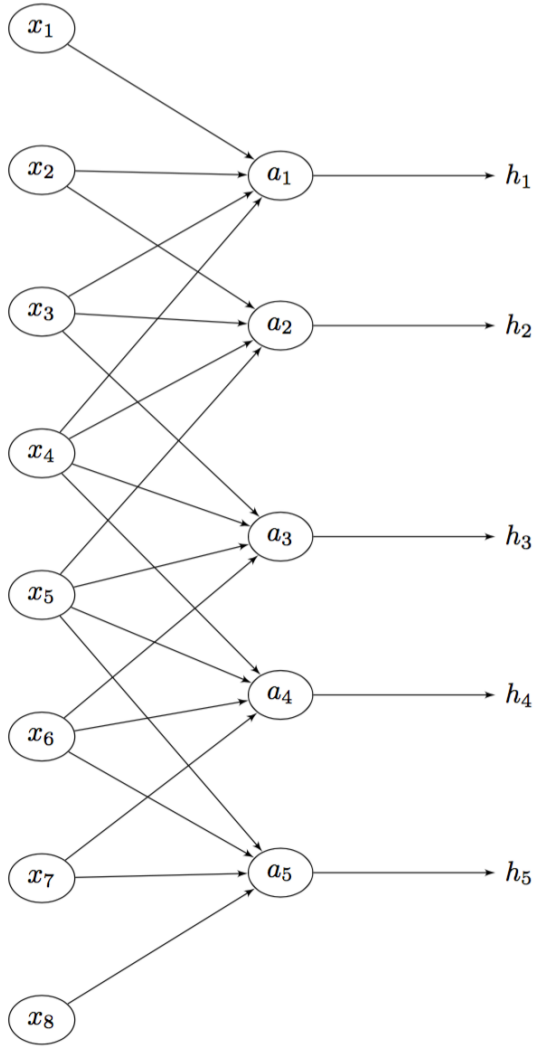
One-Dimensional Convolution



One-Dimensional Convolution



One-Dimensional Convolution



Convolutional Layer Concepts

- Filter, Kernel, Feature Map
- Receptive Field, Filter Size
- Stride
- Padding

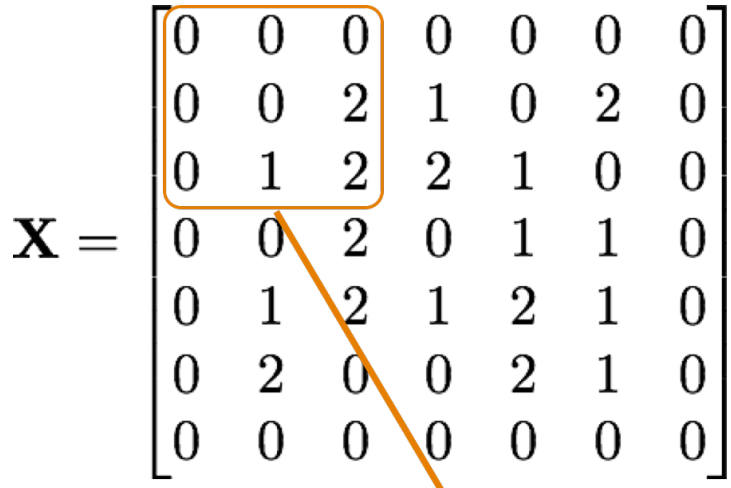
Frobenius Inner Product

$$\mathbf{X} = \begin{bmatrix} 2 & 2 & 1 \\ 2 & 0 & 1 \\ 2 & 1 & 2 \end{bmatrix}, \mathbf{K} = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix}$$

$$\begin{aligned} \langle \mathbf{X}, \mathbf{K} \rangle_F &= \sum_{i,j} x_{i,j} k_{i,j} \\ &= (2 * 1) + (2 * 1) + (1 * 1) + (2 * -1) + (0 * 1) + (1 * -1) + \\ &(2 * -1) + (1 * 1) + (2 * 1) \\ &= 2 + 2 + 1 - 2 + 0 - 1 - 2 + 1 + 2 \\ &= 3 \end{aligned}$$

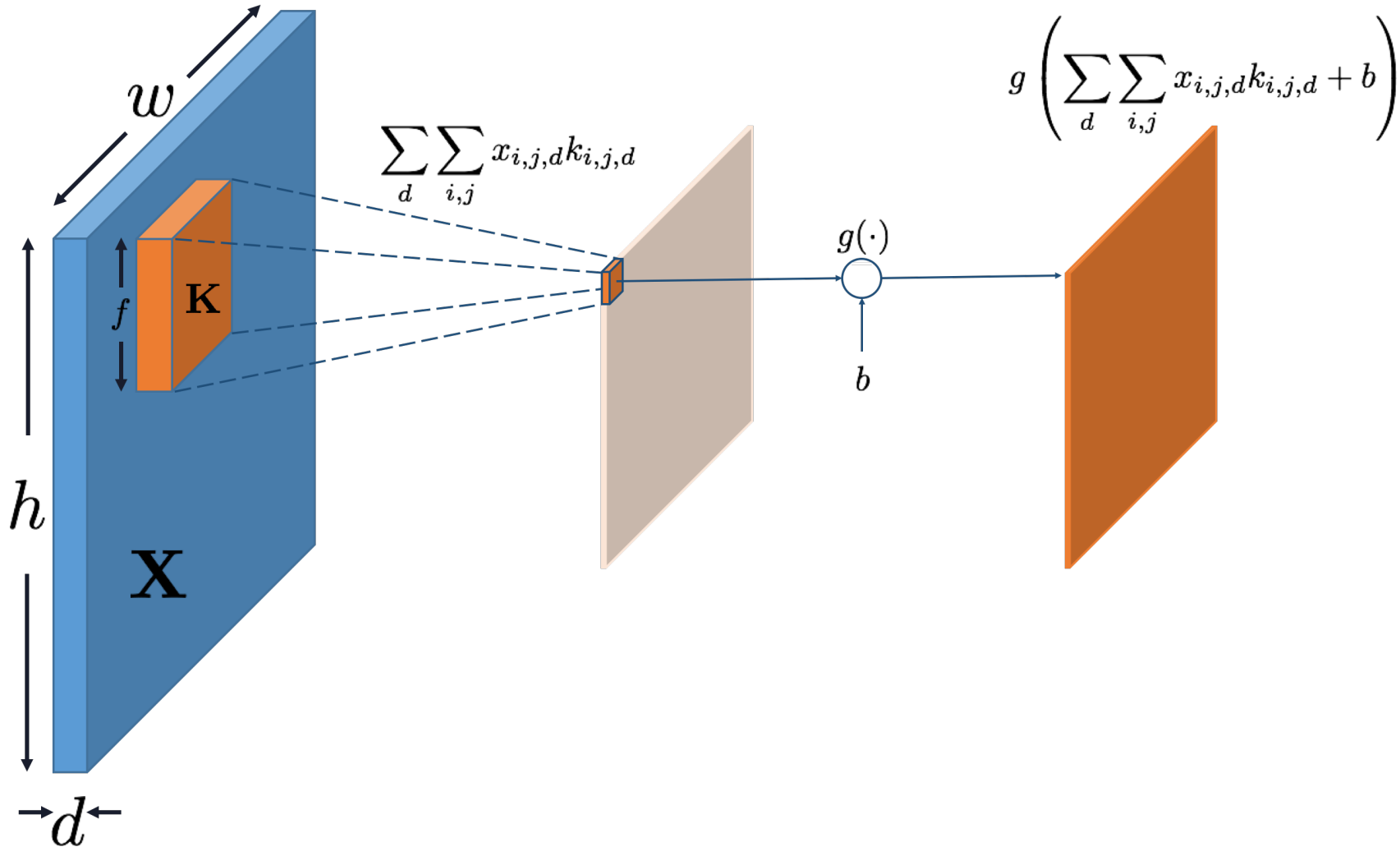
Two-Dimensional Convolution

$$\mathbf{X} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 & 2 & 0 \\ 0 & 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 & 1 & 0 \\ 0 & 1 & 2 & 1 & 2 & 1 & 0 \\ 0 & 2 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{K} = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & -1 \\ -1 & 1 & 1 \end{bmatrix}$$

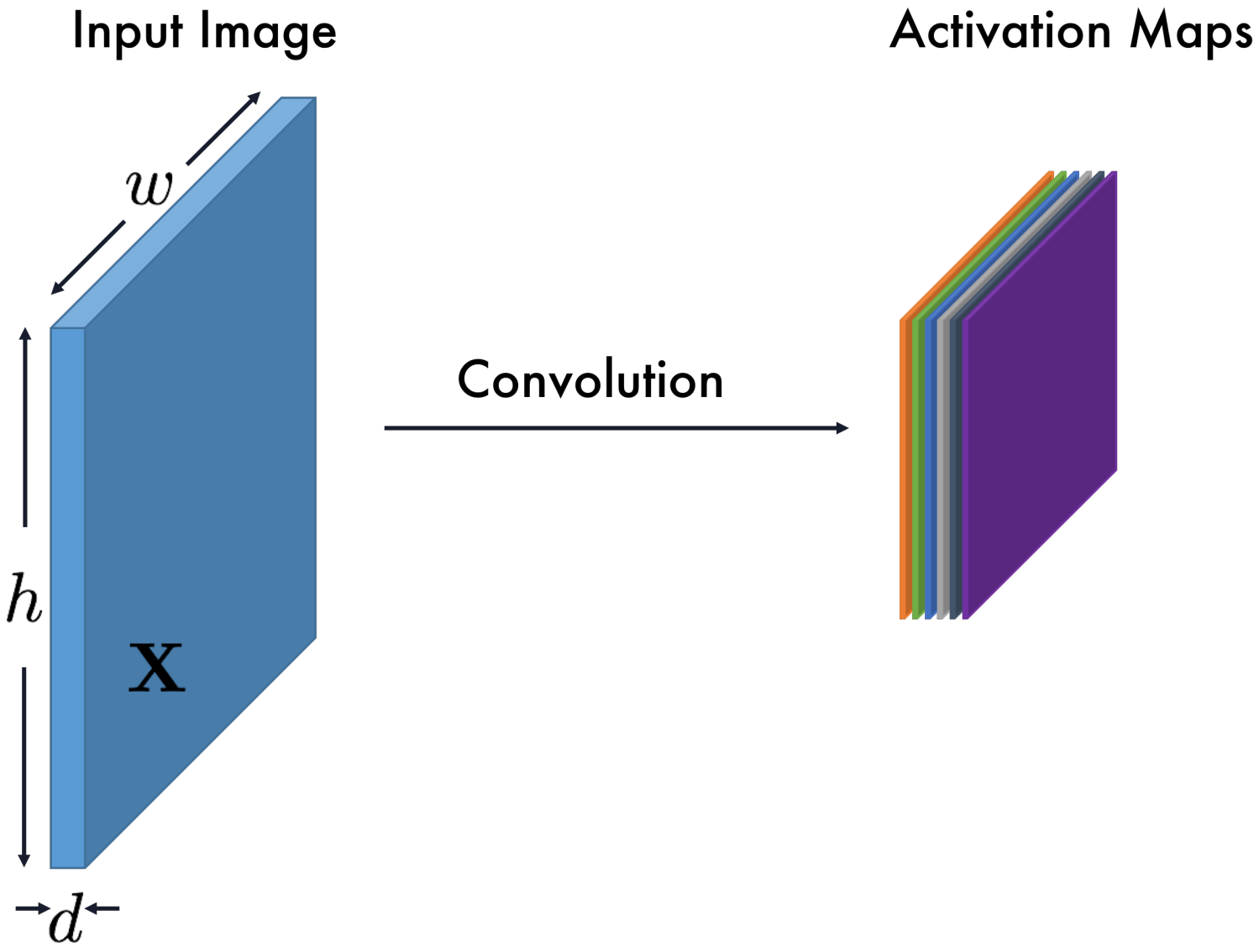


$$\mathbf{A} = \begin{bmatrix} 1 & 4 & 0 & -4 & 1 \\ 3 & 4 & 1 & 4 & 1 \\ 4 & 9 & 3 & 5 & 0 \\ 3 & 0 & 2 & 5 & 0 \\ 5 & 2 & 3 & 5 & 2 \end{bmatrix}$$

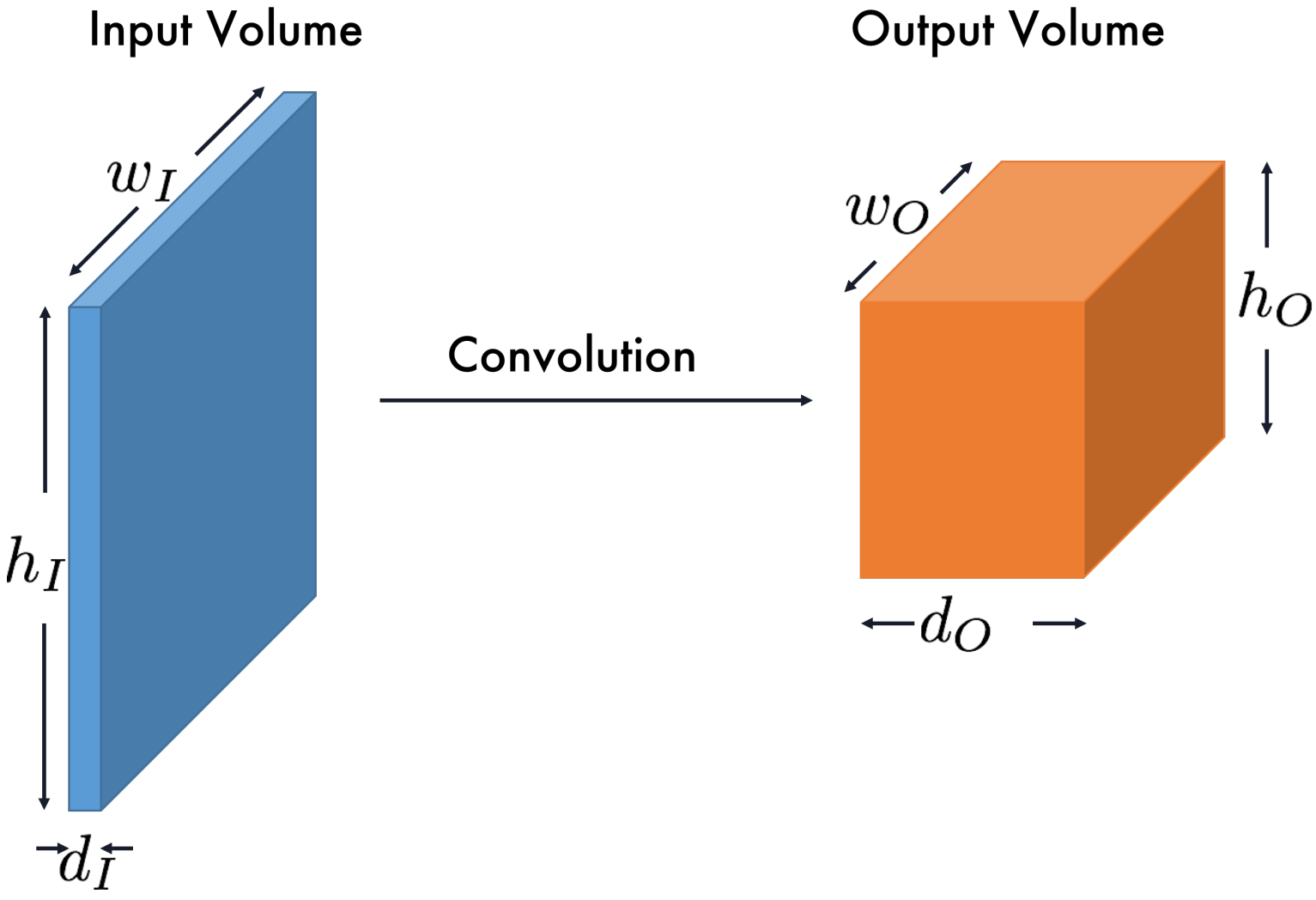
Two-Dimensional Convolution



Two-Dimensional Convolution



Two-Dimensional Convolution



Input Volume to Output Volume

For receptive field (f), stride (s), padding (p), and number of filters (m):

$$w_O = \frac{w_I - f + 2p}{s} + 1$$
$$h_O = \frac{w_I - f + 2p}{s} + 1$$
$$d_O = m$$

Number of Parameters and Multiplies

We know that a two-dimensional convolutional layer takes an input volume and produces an output volume. Let compare the number of parameters and number of multiplies for a fully-connected layer and a convolutional layer with equivalent number of elements.

- **Input Volume**: $32 \times 32 \times 3$ (3072 elements)
- **Output Volume**: $28 \times 28 \times 6$ (4704 elements)

Fully-Connected Layer

- *Every* input element is connected to *every* output element

$$\begin{aligned}\text{Parameters} &= \text{input elements} \times \text{output elements} \\ &= (32 \times 32 \times 3) \times (28 \times 28 \times 6) \\ &= 3072 \times 4704 = 14,450,688\end{aligned}$$

$$\begin{aligned}\text{Multiples} &= \text{parameters} \\ &= 4,450,688\end{aligned}$$

Convolutional Layer

- *Not* every input element is connected to *every* output element
- Number of parameters is just total number of filter parameters

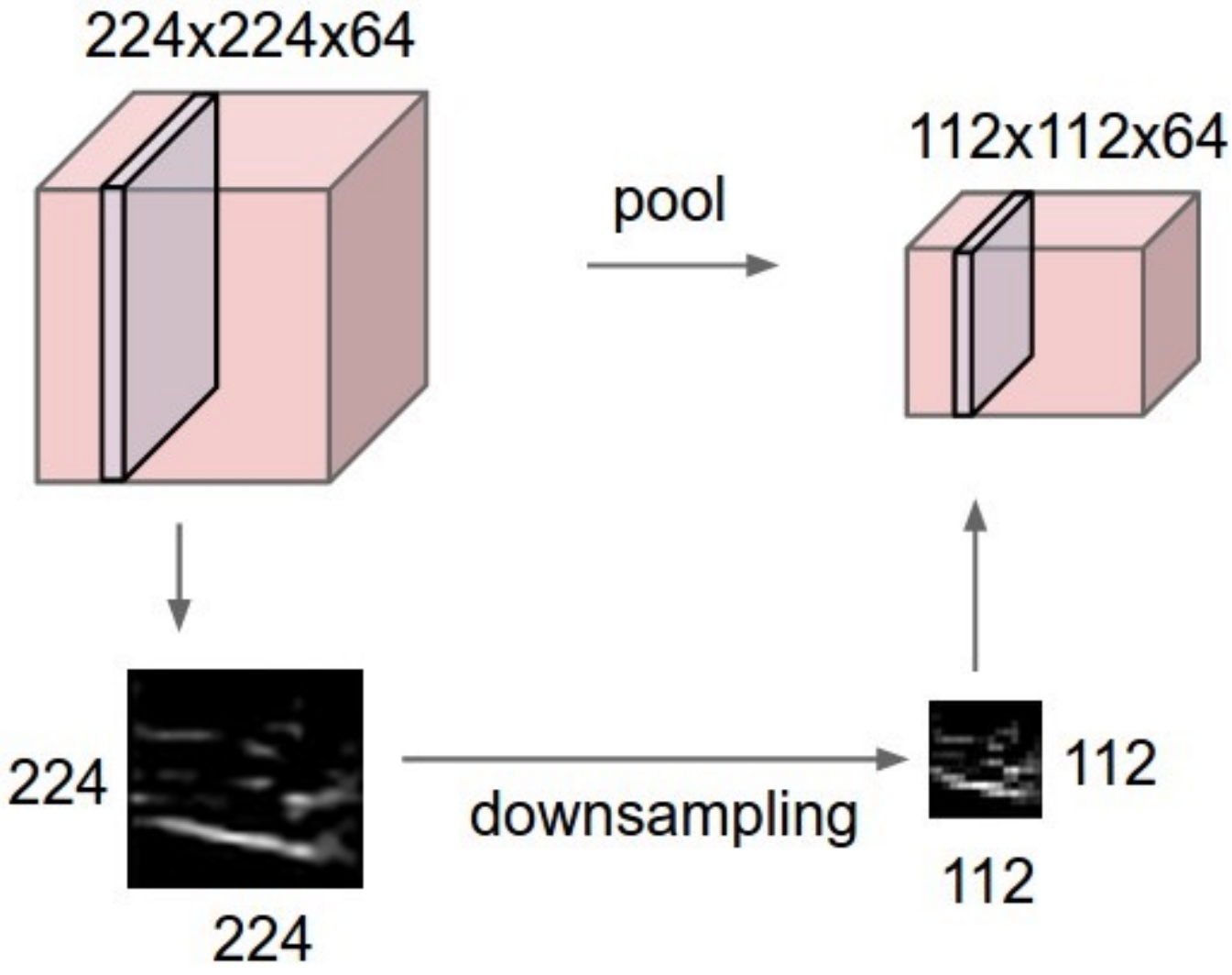
$$\begin{aligned}\text{Parameters} &= \text{num. filters} \times \text{filter elements} \\ &= 6 \times (5 \times 5 \times 3) = 450\end{aligned}$$

- *Every* output element is created by a multiply of filter elements

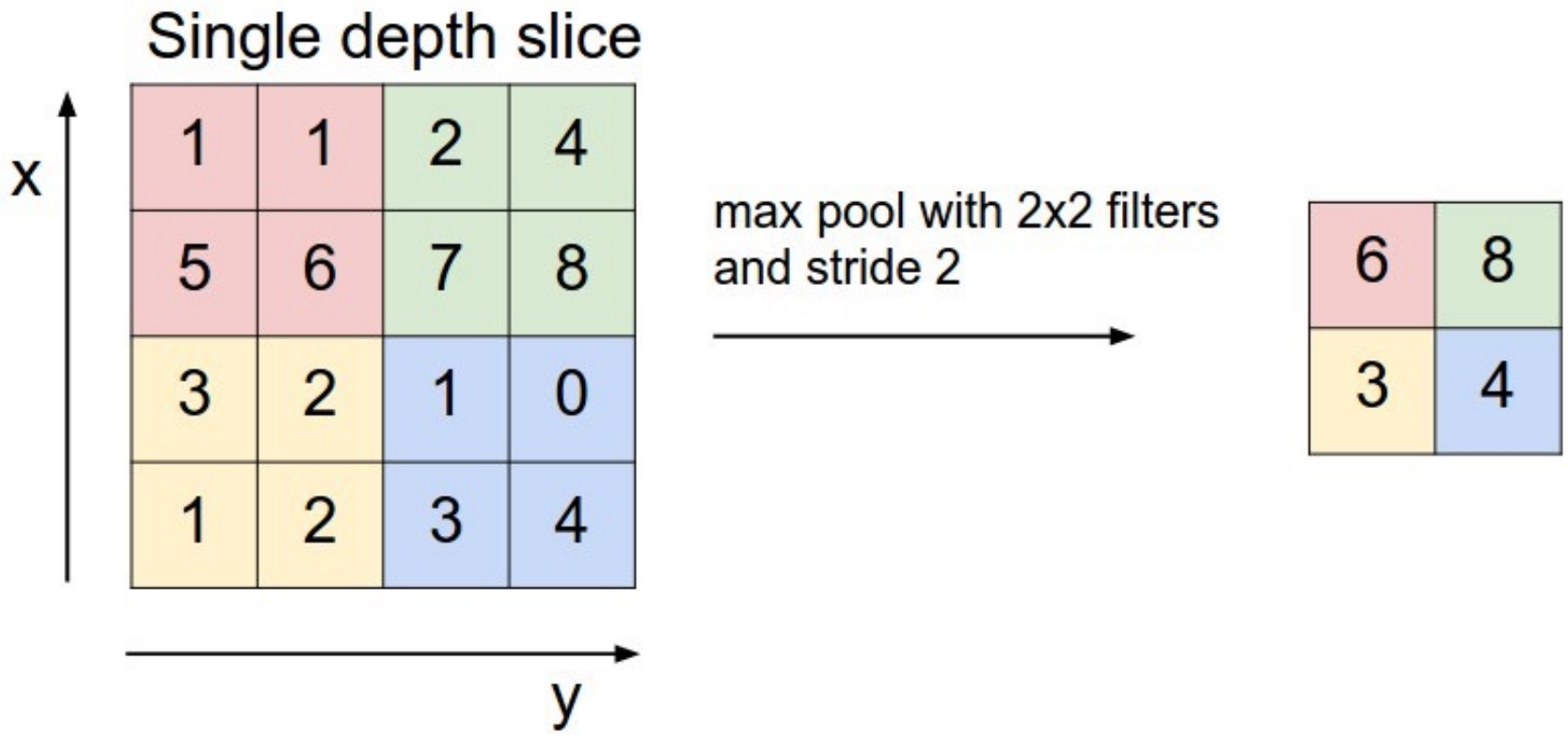
$$\begin{aligned}\text{Multiples} &= \text{output elements} \times \text{filter elements} \\ &= (28 \times 28 \times 6) \times (5 \times 5 \times 3) \\ &= 4704 \times 75 = 352,800\end{aligned}$$

- The fully connected layer has close to a 1000x more parameters and over 12x more multiplies.

Pooling



Max Pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Outline

LeNet-5 Architecture

PROC. OF THE IEEE, NOVEMBER 1998

Gradient-Based Learning Applied to Document Recognition

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner

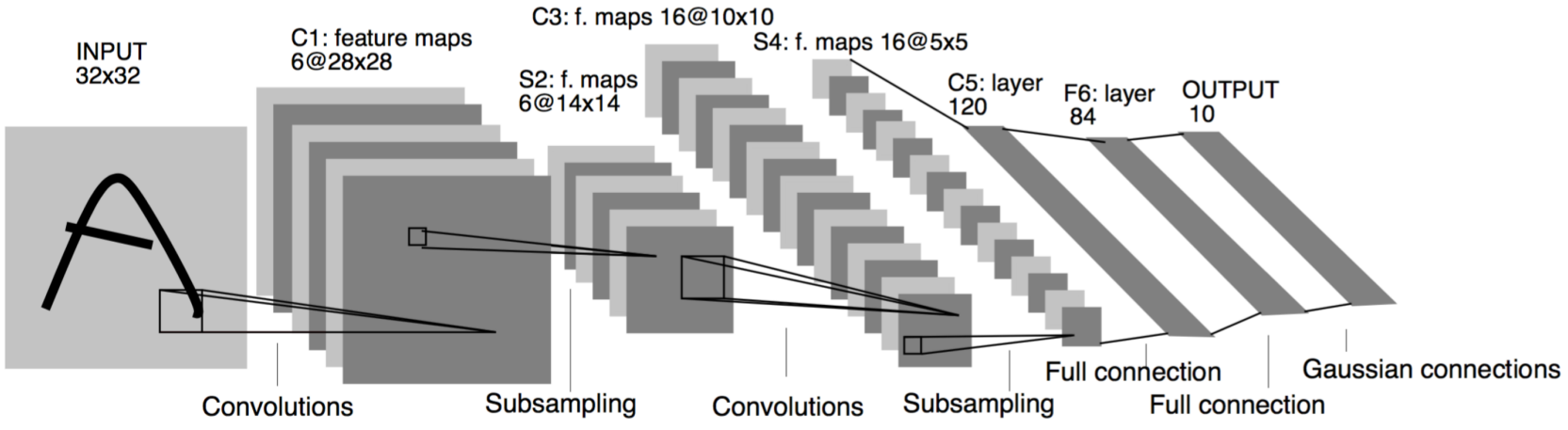


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

LeNet Case Study

"Convolutional Networks combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: **local receptive fields**, **shared weights** (or weight replication), and spatial or temporal **sub-sampling**."

"With local receptive fields, neurons can extract elementary visual features such as oriented edges, end-points, corners (or similar features in other signals such as speech spectrograms). These features are then combined by the subsequent layers in order to detect higher-order features."

"Elementary feature detectors that are useful on one part of the image are likely to be useful across the entire image. This knowledge can be applied by forcing a set of units, whose receptive fields are located at different places on the image, to have identical weight vectors."

LeNet Case Study (cont.)

"Units in a layer are organized in planes within which all the units share the same set of weights. The set of outputs of the units in such a plance is called a **feature map**. Units in a feature map are all constrained to perfrom the same operation on different parts of the image."

"A complete convolutional layer is composed of several feature maps (with different weight vectors), so that multiple features can be extracted at each location."

"The kernel of the convolution is the set of connection weights used by the units in the feature."

LeNet Case Study (cont.)

"Once a feature has been detected, its exact location becomes less important. Only its approximate position relative to other features is relevant. Only its approximate position relative to other features is relevant."

"For example, once we know that the input image contains the endpoint of a roughly horizontal segment in the upper left area, a corner in the upper right area, and the endpoint of a roughly vertical segment in the lower portion of the image, we can tell the input image is a 7. Not only is the precise position of each of those features irrelevant for identifying the pattern, it is potentially harmful because the positions are likely to vary for different instances of the character."

LeNet Case Study (cont.)

"A simple way to reduce the precision in with which the position of distinctive features are encoded in a feature map is to reduce the spatial resolution of the feature map. This can be achieved with a so-called **sub-sampling layers** which performs a local averaging and a sub-sampling, reducing the resolution of the feature map, and reducing the sensitivity of the output to shifts and distortions."

LeNet-5 Architecture

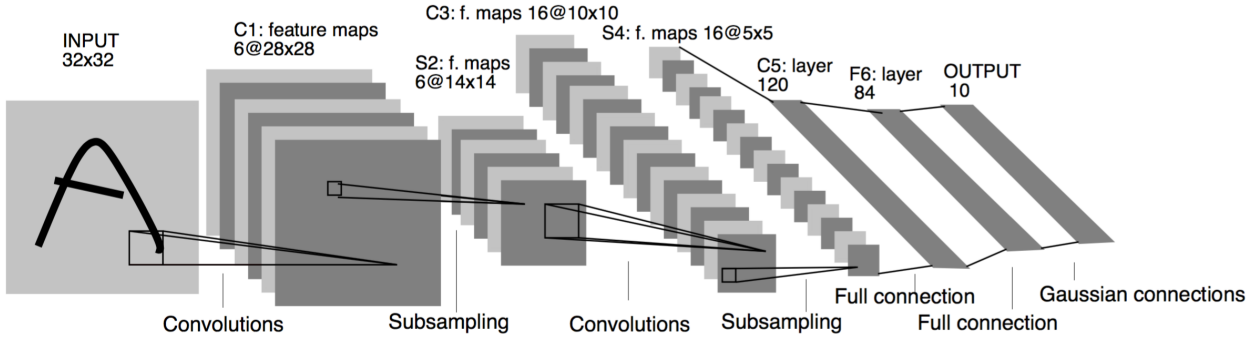
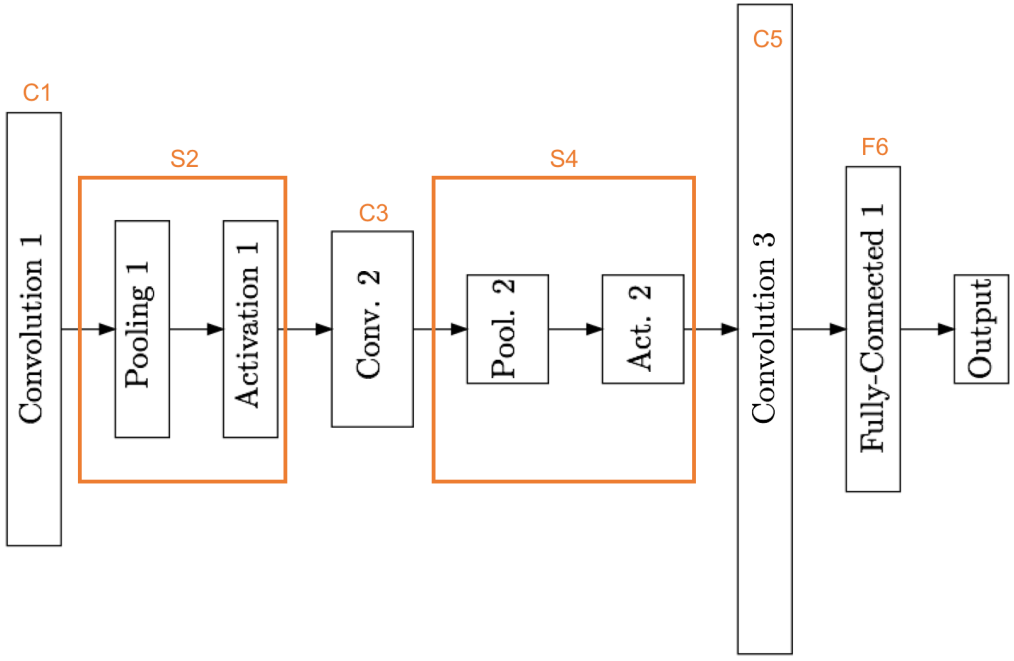


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.



Convolution Layer 1

Layer C1 is a convolutional layer with 6 feature maps. Each unit in each feature map is connected to a 5x5 neighborhood in the input. The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. C1 contains 156 trainable parameters, and 122,304 connections.

Convolution 1 (C1)

- **Input**: 32 x 32 x 1 (1024 units)
- **Convolution 1**: $f = 5$, $m = 6$, $s = 1$ (28 x 28 x 6)

Layer C1 is a convolutional layer with 6 feature maps. Each unit in each feature map is connected to a 5x5 neighborhood in the input. The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. C1 contains 156 trainable parameters, and 122,304 connections.

$$W_O = H_O = (32 - 5) + 1 = 28$$
$$D_O = 6$$

$$\text{Output volume} = (28 \times 28 \times 6) = 4704$$

$$\begin{aligned}\text{Parameters} &= \text{filter parameters} + \text{bias} \\ &= (5 \times 5 \times 6) + 6 = 150 + 6 \\ &= 156\end{aligned}$$

$$\begin{aligned}\text{Connections} &= \text{output elements} \times \text{parameters} \\ &= 4704 \times ((5 \times 5) + 1) \\ &= 4704 \times 26 = 122,304\end{aligned}$$

Pooling 1 (S2)

Layer S2 is a sub-sampling layer with 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C1. The four inputs to a unit in S2 are added, then multiplied by a trainable coefficient, and added to a trainable bias. The result is passed through a sigmoidal function. The 2x2 receptive fields are non-overlapping, therefore feature maps in S2 have half the number of rows and column as feature maps in C1. Layer S2 has 12 trainable parameters and 5,880 connections.

Pooling 1 (S2)

- **Pooling 1**: $f = 2,\ m = 6,\ s = 2$ (14 x 14 x 6)
- **Activation 1**: Sigmoid

Layer S2 is a sub-sampling layer with 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C1. The four inputs to a unit in S2 are added, then multiplied by a trainable coefficient, and added to a trainable bias. The result is passed through a sigmoidal function. The 2x2 receptive fields are non-overlapping, therefore feature maps in S2 have half the number of rows and column as feature maps in C1. Layer S2 has 12 trainable parameters and 5,880 connections.

$$W_O = H_O = \frac{(28-2)}{2} + 1 = 14$$

$$D_O = 6$$

$$\text{Output volume} = (14 \times 14 \times 6) = 1176$$

$$\text{Parameters} = \text{filter parameters} + \text{bias}$$
$$= 6 + 6 = 12$$

$$\text{Connections} = \text{output elements} \times \text{parameters}$$
$$= 1176 \times ((2 \times 2) + 1)$$
$$= 1176 \times 5 = 5,880$$

Convolution 2 (C3)

- **Convolution 2**: $f = 5,\ m = 16,\ s = 1$ (10 x 10 x 16)

Layer C3 is a convolutional layer with 16 feature maps. Each unit in each feature map is connected to several 5x5 neighborhoods at identical locations in a subset of S2's feature maps. Table I shows the set of S2 feature maps

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

combined by each C3 feature map. Why not connect every S2 feature map to every C3 feature map? The reason is twofold. First, a non-complete connection scheme keeps the number of connections within reasonable bounds. More importantly, it forces a break of symmetry in the network. Different feature maps are forced to extract different (hopefully complementary) features because they get different sets of inputs. The rationale behind the connection scheme in table I is the following. The first six C3 feature maps take inputs from every contiguous subsets of three feature maps in S2. The next six take input from every contiguous subset of four. The next three take input from some discontinuous subsets of four. Finally the last one takes input from all S2 feature maps. Layer C3 has 1,516 trainable parameters and 151,600 connections.

- Not all "channels" of previous layer is connected to every filter
  - **Set 1**: 6 channels are connected to 3 filters
  - **Set 2**: 9 channels are connected to 4 filters
  - **Set 3**: 1 channel is connected to 5 filters

Pooling 2 (S4)

- **Pooling 2**: $f = 2$, $m = 16$, $s = 2$ (5 x 5 x 16)
- **Activation 2**: Sigmoid

Layer S4 is a sub-sampling layer with 16 feature maps of size 5x5. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C3, in a similar way as C1 and S2. Layer S4 has 32 trainable parameters and 2,000 connections.

$$W_O = H_O = \frac{(10-2)}{2} + 1 = 5$$

$$D_O = 16$$

$$\text{Output volume} = (5 \times 5 \times 16) = 400$$

$$\begin{aligned}\text{Parameters} &= \text{filter parameters} + \text{bias} \\ &= 16 + 16 = 32\end{aligned}$$

$$\begin{aligned}\text{Connections} &= \text{output elements} \times \text{parameters} \\ &= 400 \times ((2 \times 2) + 1) \\ &= 400 \times 5 = 2,000\end{aligned}$$

Convolution 3 (C5)

- **Convolution 3**: $f = 5$, $m = 120$, $s = 1$ (1 x 1 x 120)
- **Activation 3**: Tanh

Layer C5 is a convolutional layer with 120 feature maps. Each unit is connected to a 5x5 neighborhood on all 16 of S4's feature maps. Here, because the size of S4 is also 5x5, the size of C5's feature maps is 1x1: this amounts to a full connection between S4 and C5. C5 is labeled as a convolutional layer, instead of a fully-connected layer, because if LeNet-5 input were made bigger with everything else kept constant, the feature map dimension would be larger than 1x1. This process of dynamically increasing the size of a convolutional network is described in the section Section VII. Layer C5 has 48,120 trainable connections.

$$W_O = H_O = (5 - 5) + 1 = 1$$
$$D_O = 120$$

$$\text{Output volume} = (1 \times 1 \times 120) = 120$$

$$\begin{aligned}\text{Connections} &= \text{output elements} \times \text{parameters} \\ &= 120 \times ((20 \times 20) + 1) \\ &= 120 \times 401 = 48,120\end{aligned}$$

Fully-Connected 1 (F6)

Layer F6, contains 84 units (the reason for this number comes from the design of the output layer, explained below) and is fully connected to C5. It has 10,164 trainable parameters.

Fully-Connected 1 (F6)

- **Fully-Connected 1**: 84

Layer F6, contains 84 units (the reason for this number comes from the design of the output layer, explained below) and is fully connected to C5. It has 10,164 trainable parameters.

$$\begin{aligned}\text{Parameters} &= \text{Connections} \\ &= \text{input elements} \times \text{output elements} \\ &= 120 \times 84 + 84 \\ &= 10,164\end{aligned}$$

LeNet Case Study

"Successive layers of convolutions and sub-sampling are typically alternated, resulting in a "bi-pyramid": at each layer, the number of feature maps is increased as the spatial resolution is decreased."

"The weight sharing technique has the interesting side effect of reducing the "capacity" of the machine and reducing the gap between test error and training error."

"LeNet-5 contains 340,908 connections, but only 60,000 trainable free parameters because of the weight sharing."

LeNet-5 Architecture

- **Input**: 32 x 32 x 1 (1024 units)
- **Convolution 1**: $f = 5$, $m = 6$, $s = 1$ (28 x 28 x 6)
- **Pooling 1**: $f = 2$, $m = 6$, $s = 2$ (14 x 14 x 6)
- **Activation 1**: Sigmoid
- **Convolution 2**: $f = 5$, $m = 16$, $s = 1$ (10 x 10 x 16)
- **Pooling 2**: $f = 2$, $m = 16$, $s = 2$ (5 x 5 x 16)
- **Activation 2**: Sigmoid
- **Convolution 3**: $f = 5$, $m = 120$, $s = 1$ (1 x 1 x 120)
- **Activation 3**: Tanh
- **Fully-Connected 1**: 84
- **Output**: 10

Outline

CNNs in Keras

- Convolutional Layers: `Conv`, `Cropping`, `UpSampling`, `ZeroPadding`
- Pooling Layers: `MaxPooling`, `AveragePooling`

Conv2D

```
keras.layers.convolutional.Conv2D(filters,
        kernel_size,
        strides=(1, 1),
        padding='valid',
        data_format=None,
        dilation_rate=(1, 1),
        activation=None,
        use_bias=True,
        kernel_initializer='glorot_uniform',
        bias_initializer='zeros',
        kernel_regularizer=None,
        bias_regularizer=None,
        activity_regularizer=None,
        kernel_constraint=None,
        bias_constraint=None)
```

MaxPooling2D

```
keras.layers.pooling.MaxPooling2D(
        pool_size=(2, 2),
        strides=None,
        padding='valid',
        data_format=None)
```

LeNet with 28x28x1 MNIST

- **Input**: 28 x 28 x 1 (784 units)
- **Convolution 1**: $f = 5$, $m = 6$, $s = 1$ (24 x 24 x 6)
- **Pooling 1**: $f = 2$, $m = 6$, $s = 2$ (12 x 12 x 6)
- **Activation 1**: Sigmoid
- **Convolution 2**: $f = 5$, $m = 16$, $s = 1$ (8 x 8 x 16)
- **Pooling 2**: $f = 2$, $m = 16$, $s = 2$ (4 x 4 x 16)
- **Activation 2**: Sigmoid
- **Convolution 3**: $f = 4$, $m = 120$, $s = 1$ (1 x 1 x 120)
- **Activation 3**: Tanh
- **Fully-Connected 1**: 84
- **Output**: 10

Outline

ImageNet

IMAGENET

ImageNet

- Over 15 million labeled high-resolution images
- 22,000 classes
- ImageNet Large Scale Visual Recognition Competition (ILSVRC)
  - Uses a subset of ImageNet: 1,000 images in each of 1,000 categories.
  - 1.2 million training images, 50,000 validation images, and 150,000 testing images.

AlexNet

- ILSVRC 2012 winner
- Breakthrough performance, achieving significant improvement from previous years

---

ImageNet Classification with Deep Convolutional Neural Networks

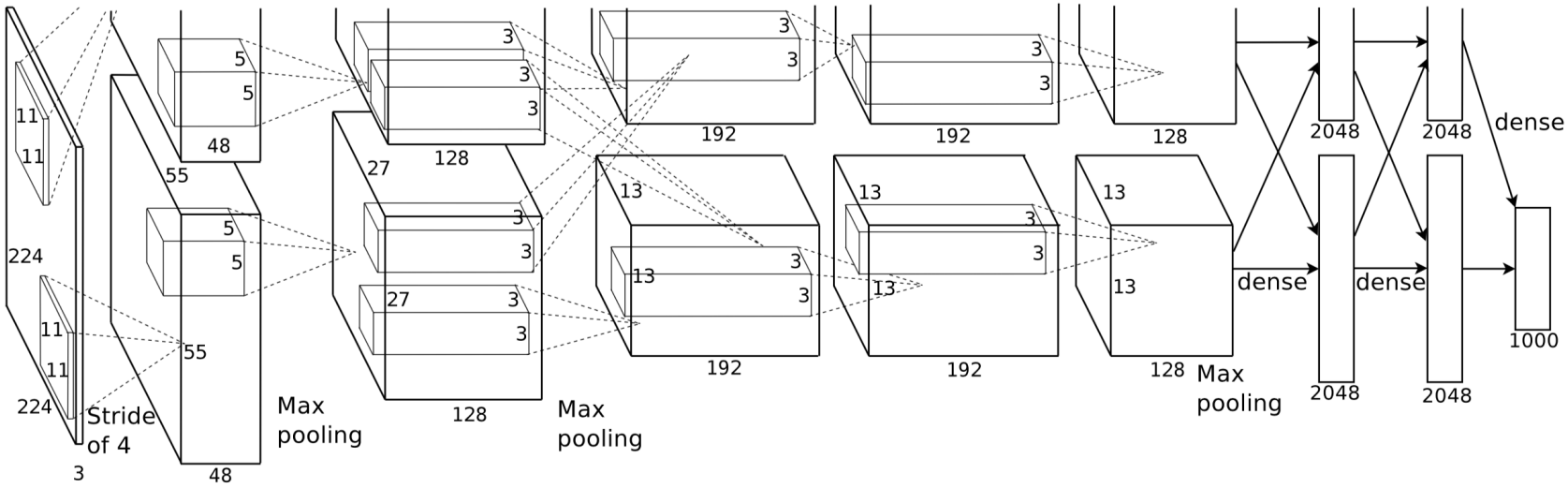
---

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

AlexNet Architecture



- Five convolutional layers, three fully-connected layers, ReLU activation
- 60 million parameters
- SGD, mini-batch of 128, 0.9 momentum, weight decay of 0.0005
- Roughly 90 epochs
- Five to six days to train on two GTX 580 3GB GPUs.

AlexNet Architecture

- **Input**: 224 x 224 x 3 image
- **Convolution 1**: 96 filters, 11 x 11 x 3/4
- **Convolution 2**: 256 filters, 5 x 5 x 48
- **Convolution 3**: 384 filters, 3 x 3 x 256
- **Convolution 4**: 384 filters, 3 x 3 x 192
- **Convolution 5**: 256 filters, 3 x 3 x 192
- **Fully Connected Layers**: 4096 neurons
- **Output**: 1000-way softmax
- **Loss**: Multinomial logistic
- Neurons of second, fourth, and fifth convolutional layers are connected to only those neurons from the previous layer which reside on the same GPU

LeNet VS AlexNet

- **Data**
  - LeNet: 60,000 training
  - AlexNet: 1.2M training
- **Compute**
  - AlexNet: GPU, CUDA
- **Algorithm**
  - LeNet: Sigmoid
  - AlexNet: ReLU, Dropout

ZFNet

- ILSVRC 2013 winner
- Denser convolutions

Visualizing and Understanding Convolutional Neural Networks

Matthew D. Zeiler ZEILER@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University

Rob Fergus FERGUS@CS.NYU.EDU
Dept. of Computer Science, Courant Institute, New York University

ZFNet Architecture

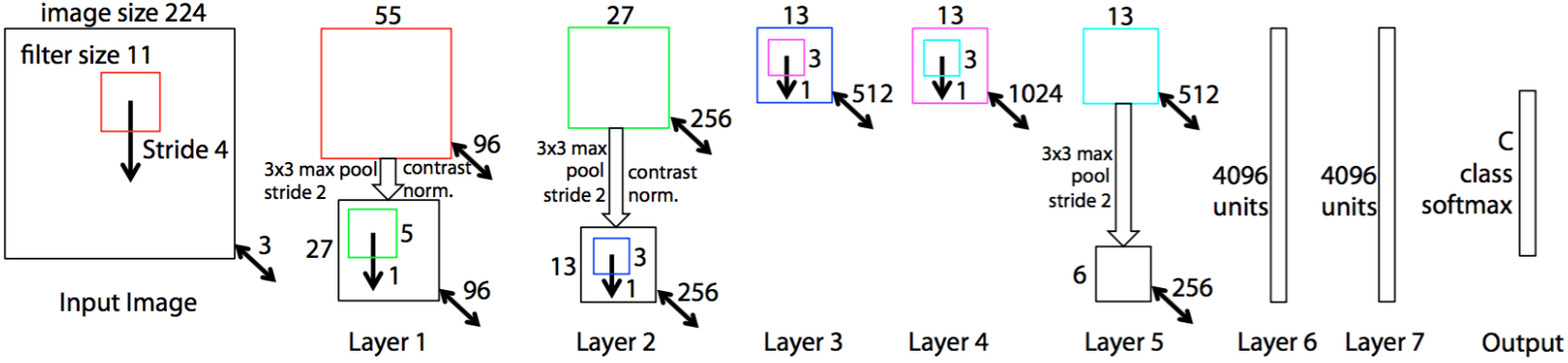


Figure 2. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 11 by 11, using a stride of 4 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 27 by 27 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

VGG

- ILSVRC 2014 second place
- Demonstrated homogeneous convolutional layers

VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

Karen Simonyan* **& Andrew Zisserman+**
Visual Geometry Group, Department of Engineering Science, University of Oxford
{karen,az}@robots.ox.ac.uk

VGG Architecture

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	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 conv3-512 **conv1-512**	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 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

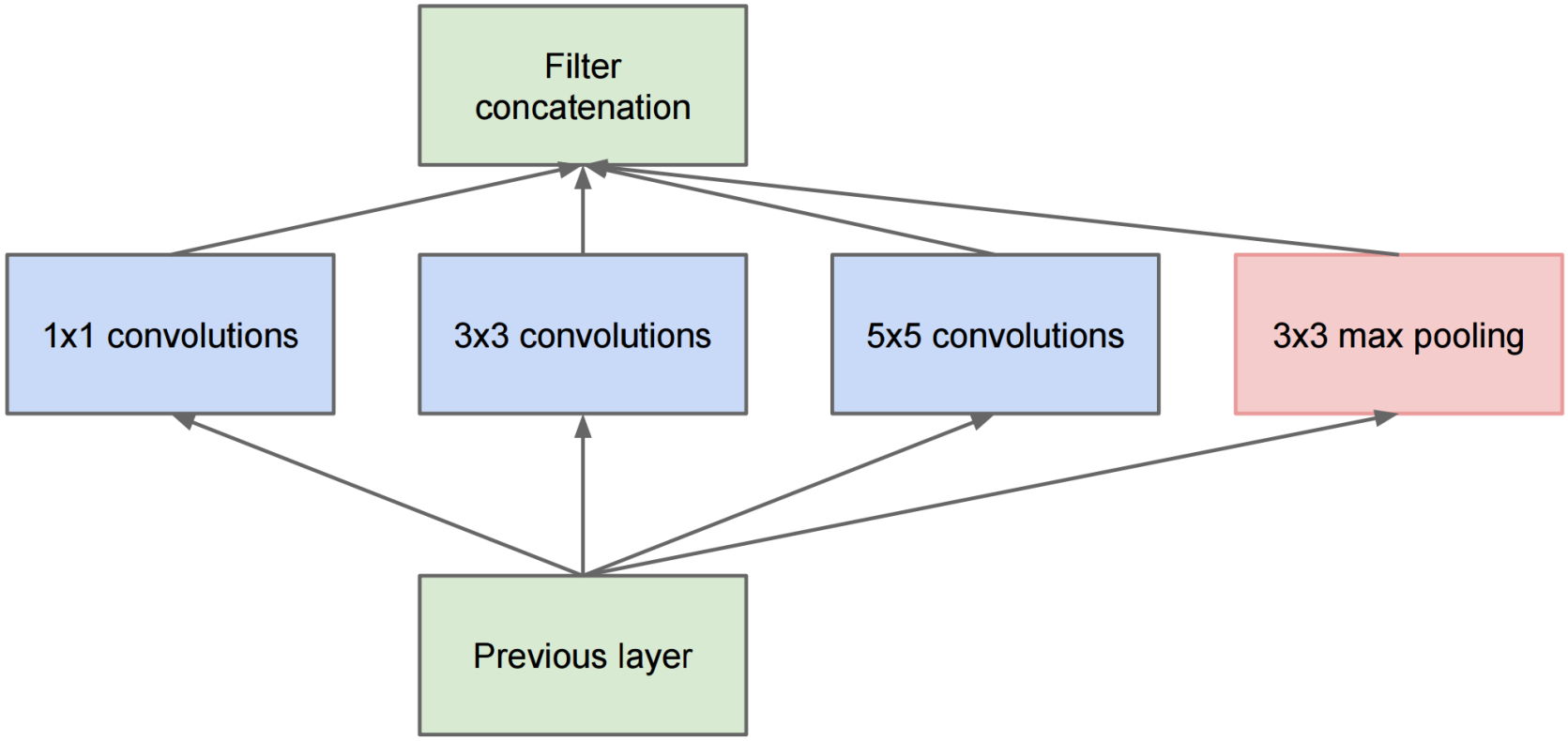
GoogLeNet

- ILSVRC 2014 winner
- Inception module

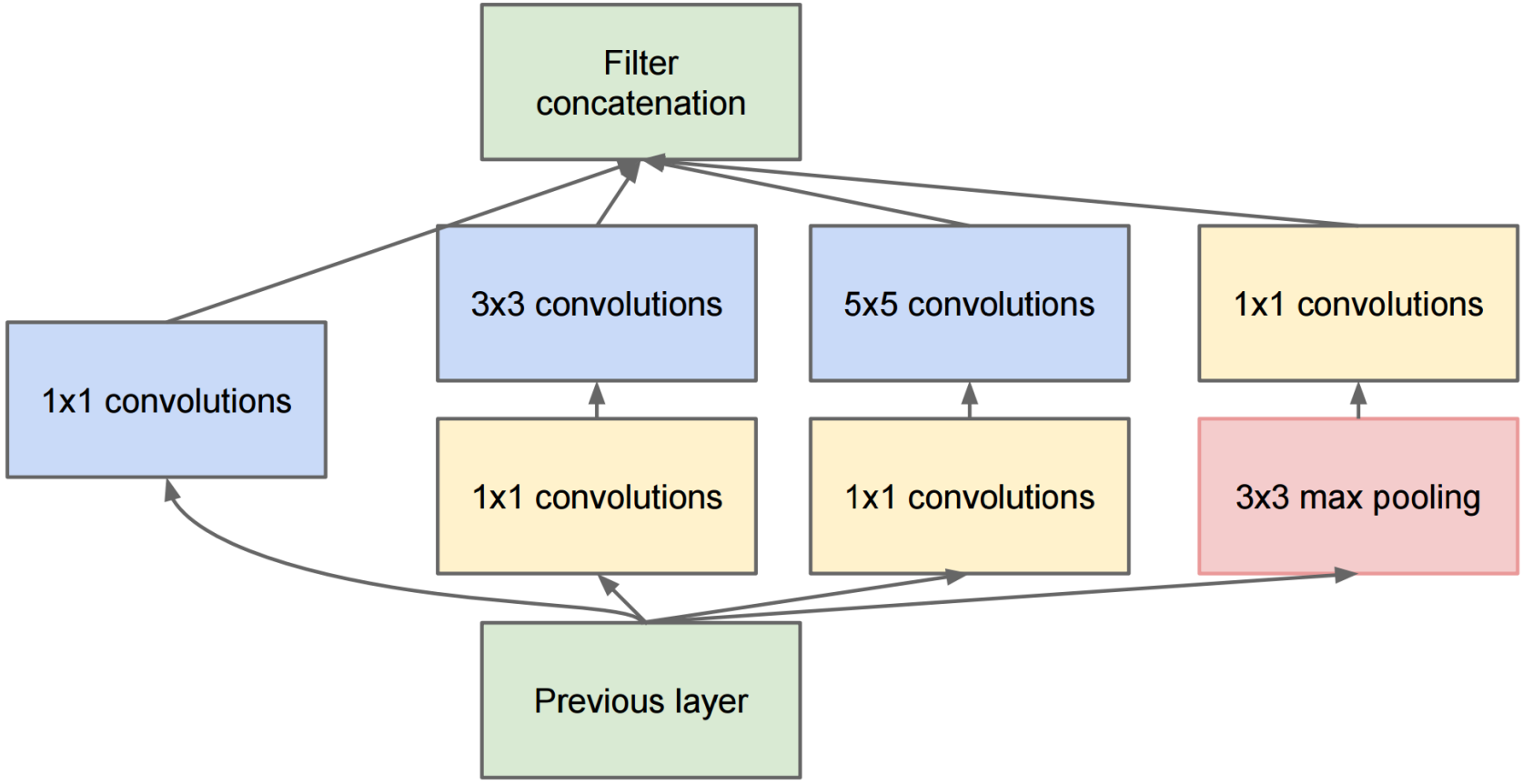
Going Deeper with Convolutions

Christian Szegedy[1], Wei Liu[2], Yangqing Jia[1], Pierre Sermanet[1], Scott Reed[3],
Dragomir Anguelov[1], Dumitru Erhan[1], Vincent Vanhoucke[1], Andrew Rabinovich[4]
[1]Google Inc. [2]University of North Carolina, Chapel Hill
[3]University of Michigan, Ann Arbor [4]Magic Leap Inc.
[1]{szegedy,jiayq,sermanet,dragomir,dumitru,vanhoucke}@google.com
[2]wliu@cs.unc.edu, [3]reedscott@umich.edu, [4]arabinovich@magicleap.com

Inception Module (Naive)



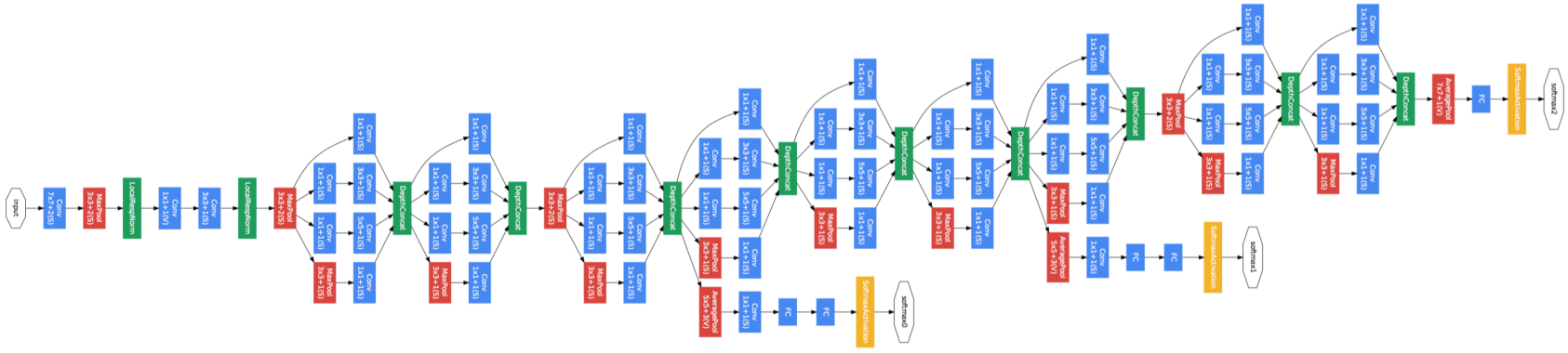
Inception Module (Dimension Reductions)



GoogLeNet Inception Architecture

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

GoogLeNet Network



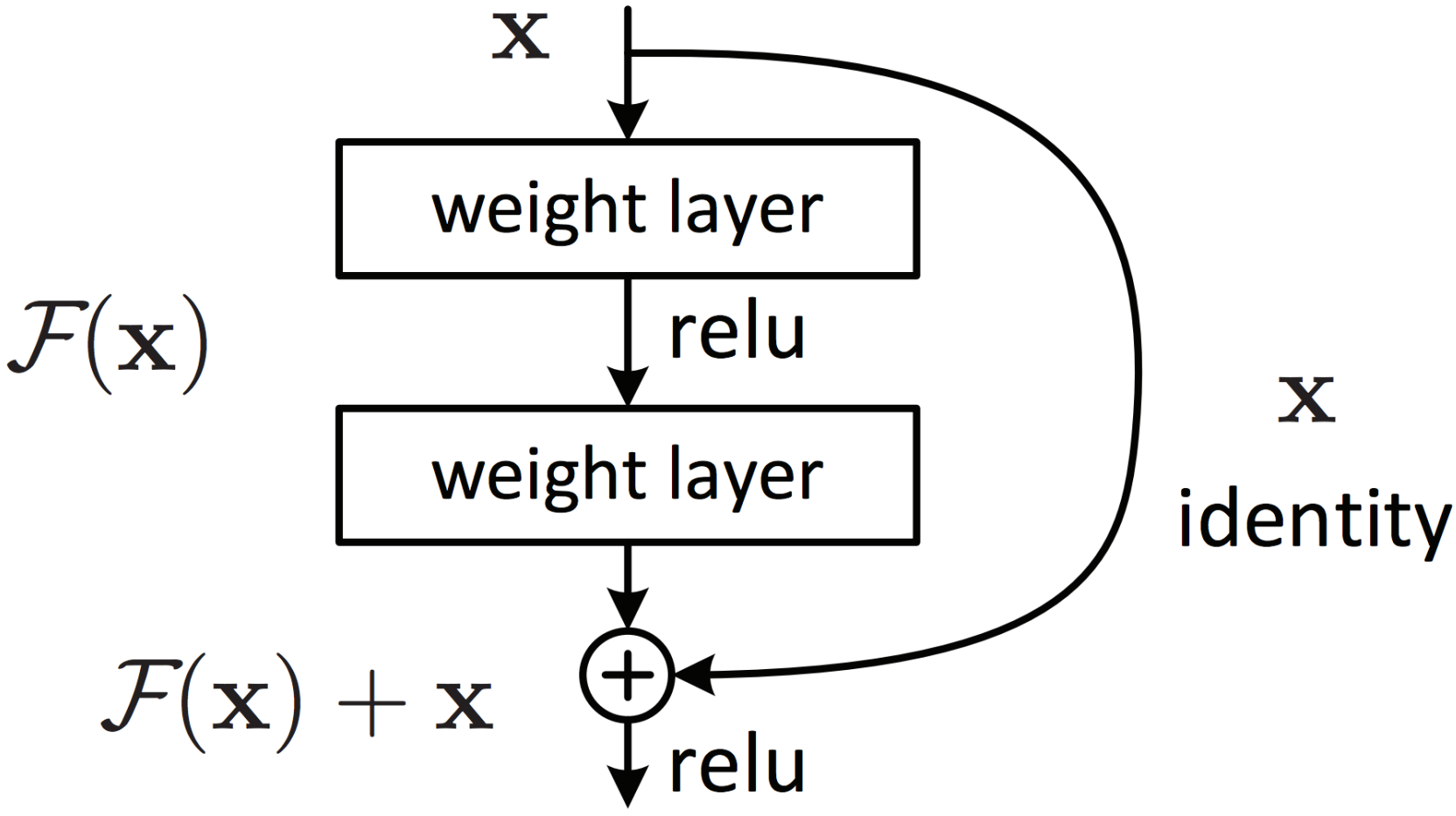
ResNet

- ILSVRC 2015 winner
- Introduced skip connections; enabled much deeper networks

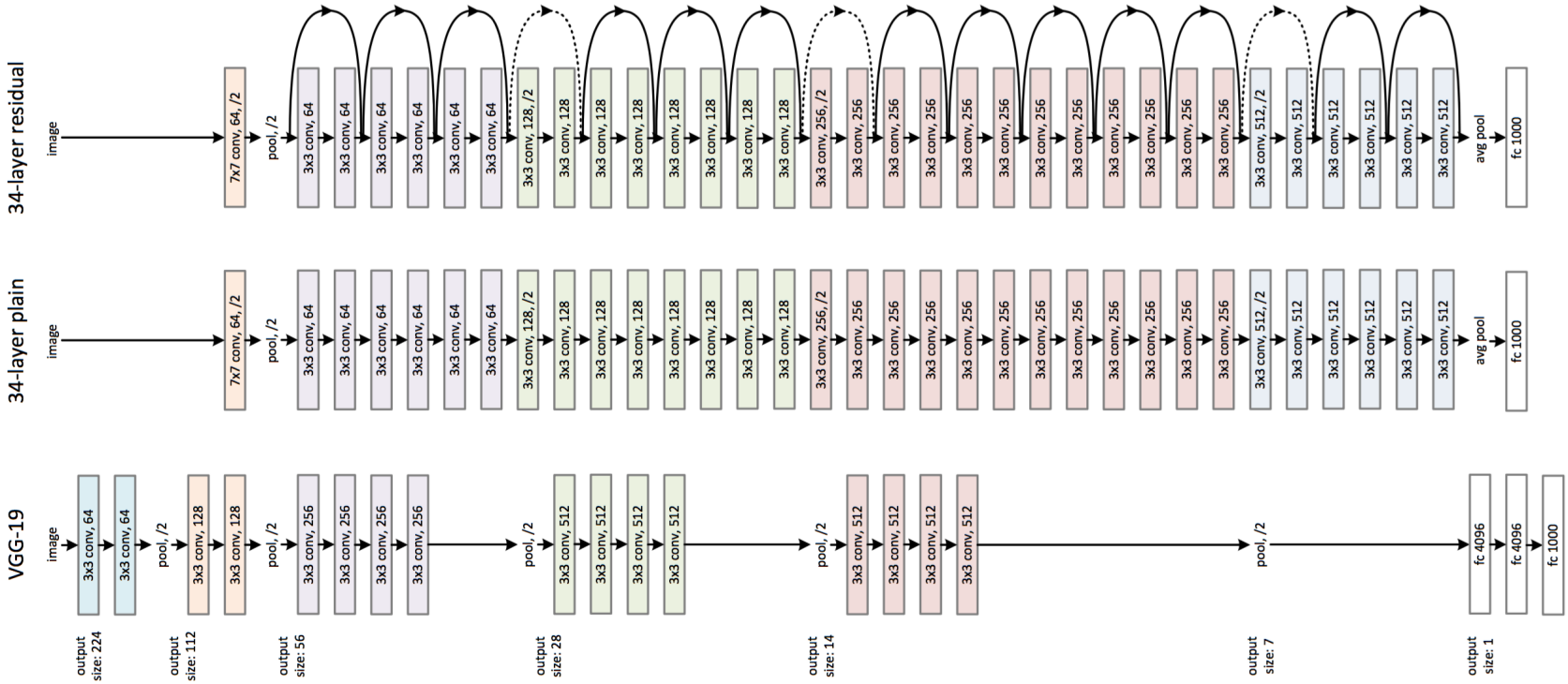
Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

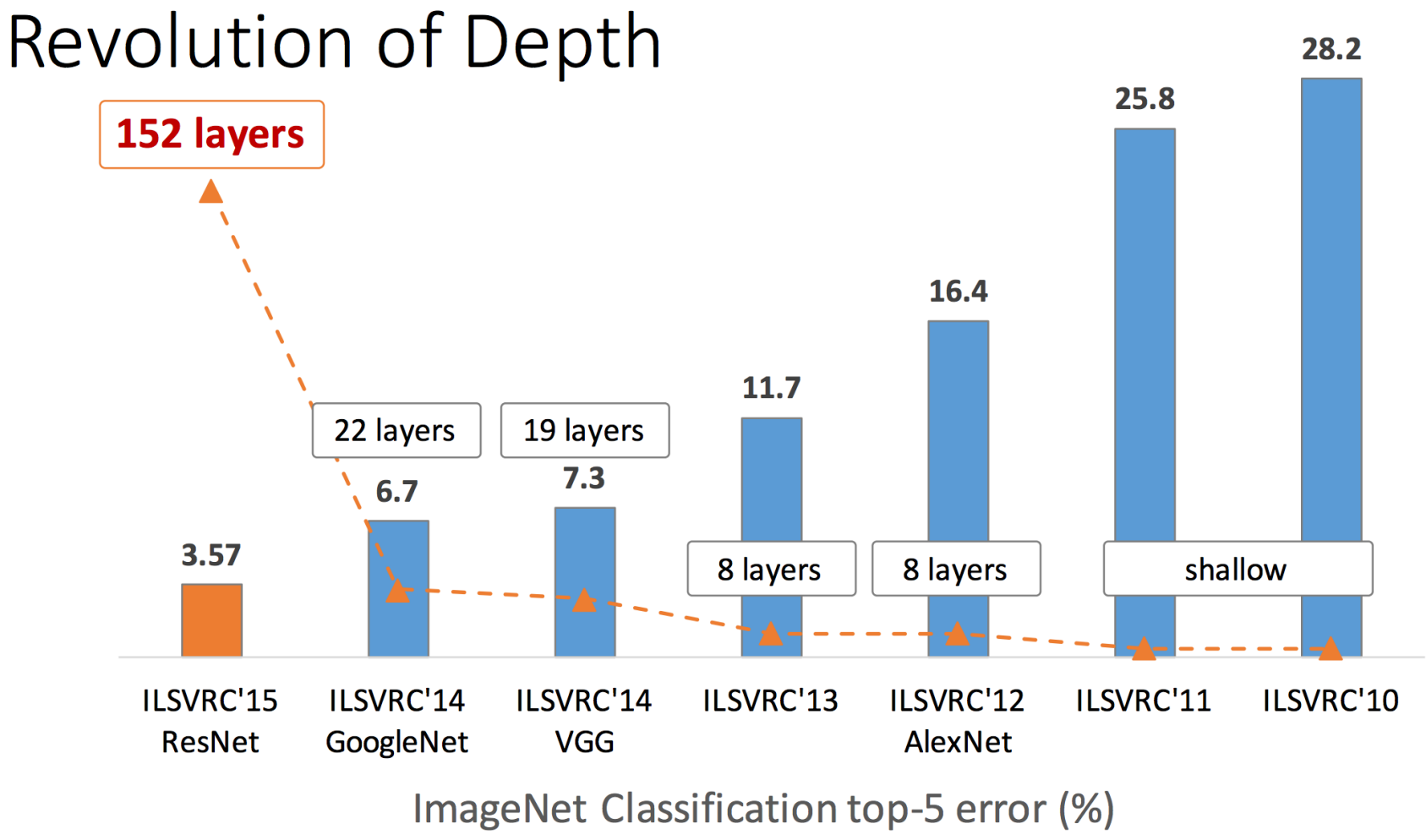
Residual Learning Building Block



Residual Network



Revolution of Depth



Inception ResNet

Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

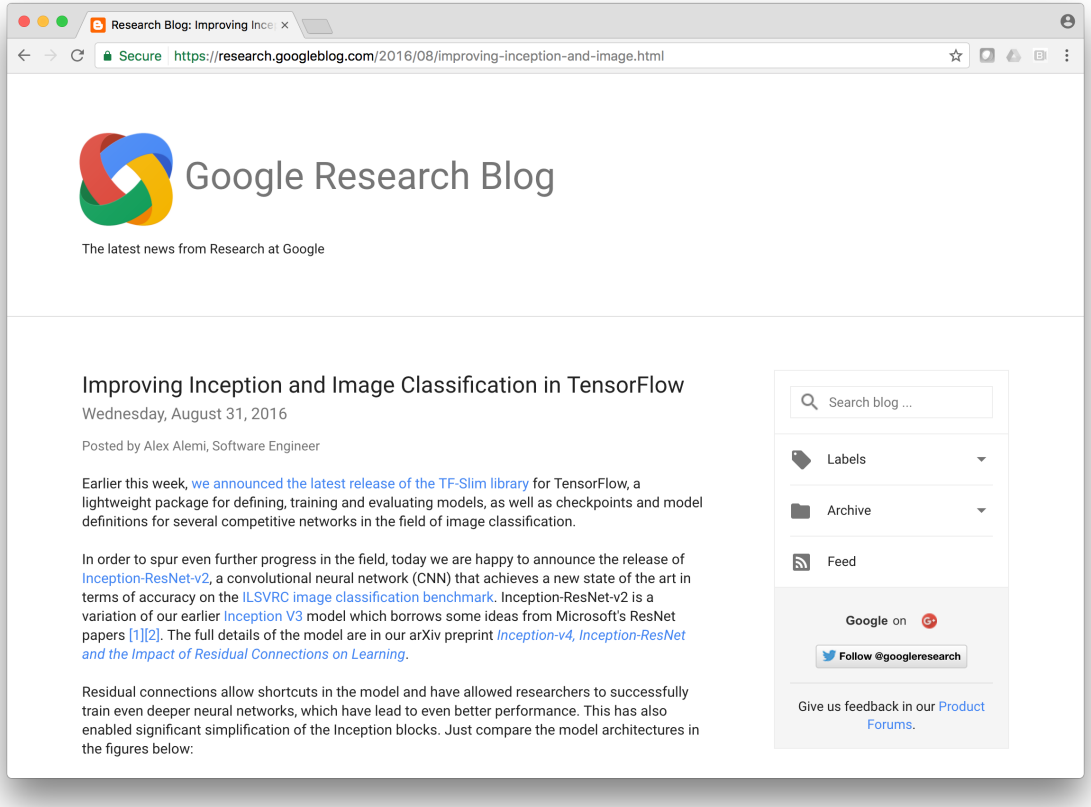
Christian Szegedy
Google Inc.
1600 Amphitheatre Pkwy, Mountain View, CA
szegedy@google.com

Sergey Ioffe
sioffe@google.com

Vincent Vanhoucke
vanhoucke@google.com

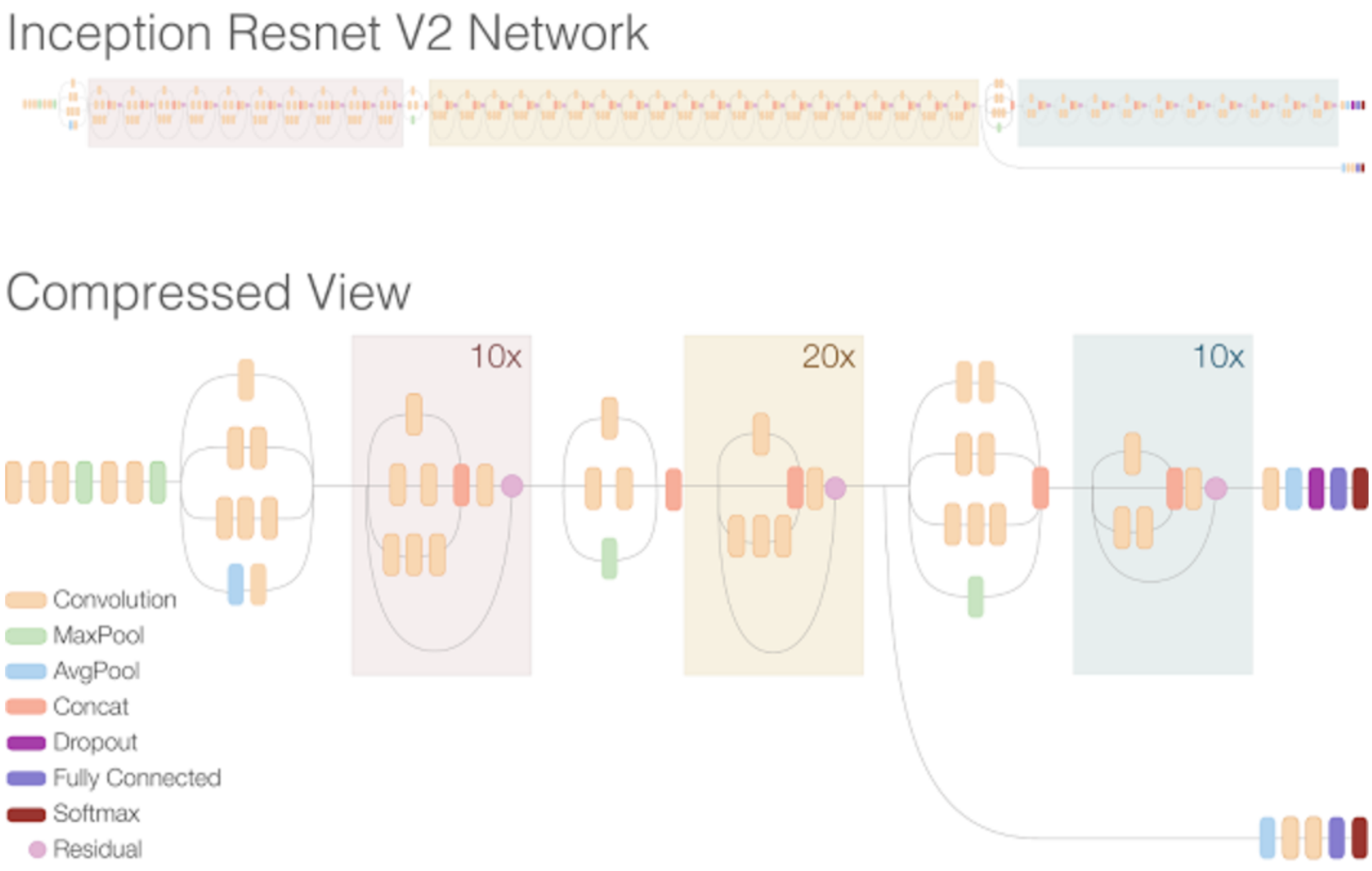
Alex Alemi
alemi@google.com

Inception ResNet



https://research.googleblog.com/2016/08/improving-inception-and-image.html

Inception ResNet V2 Architecture



Schematic diagram of Inception-ResNet-v2

Network Comparison

AN ANALYSIS OF DEEP NEURAL NETWORK MODELS FOR PRACTICAL APPLICATIONS

Alfredo Canziani & Eugenio Culurciello
Weldon School of Biomedical Engineering
Purdue University
{canziani,euge}@purdue.edu

Adam Paszke
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
a.paszke@students.mimuw.edu.pl

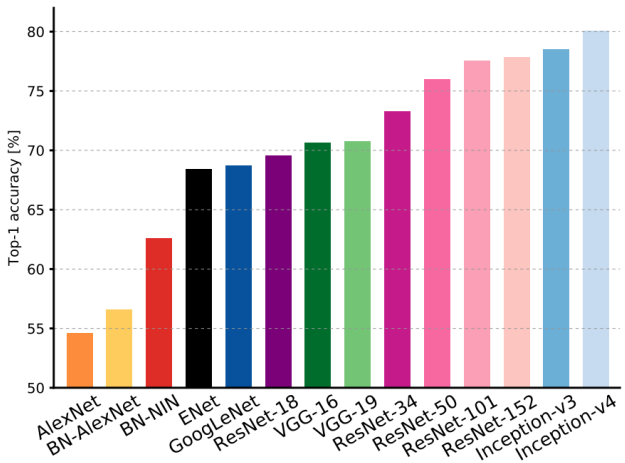


Figure 1: **Top1 *vs.* network.** Single-crop top-1 validation accuracies for top scoring single-model architectures. We introduce with this chart our choice of colour scheme, which will be used throughout this publication to distinguish effectively different architectures and their correspondent authors. Notice that networks of the same group share the same hue, for example ResNet are all variations of pink.

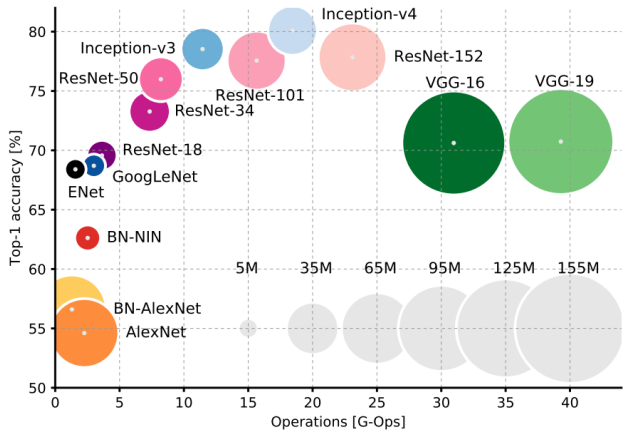
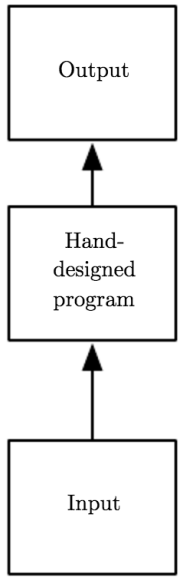


Figure 2: **Top1 *vs.* operations, size $\propto$ parameters.** Top-1 one-crop accuracy versus amount of operations required for a single forward pass. The size of the blobs is proportional to the number of network parameters; a legend is reported in the bottom right corner, spanning from 5×10^6 to 155×10^6 params. Both these figures share the same y-axis, and the grey dots highlight the centre of the blobs.

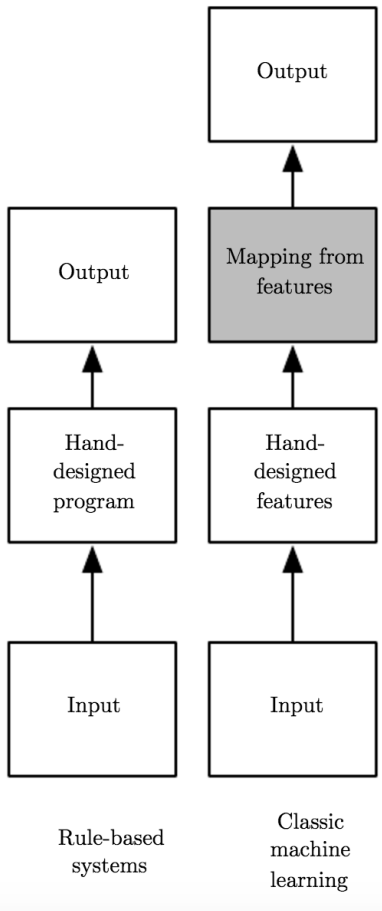
Outline

Rule-Based Systems

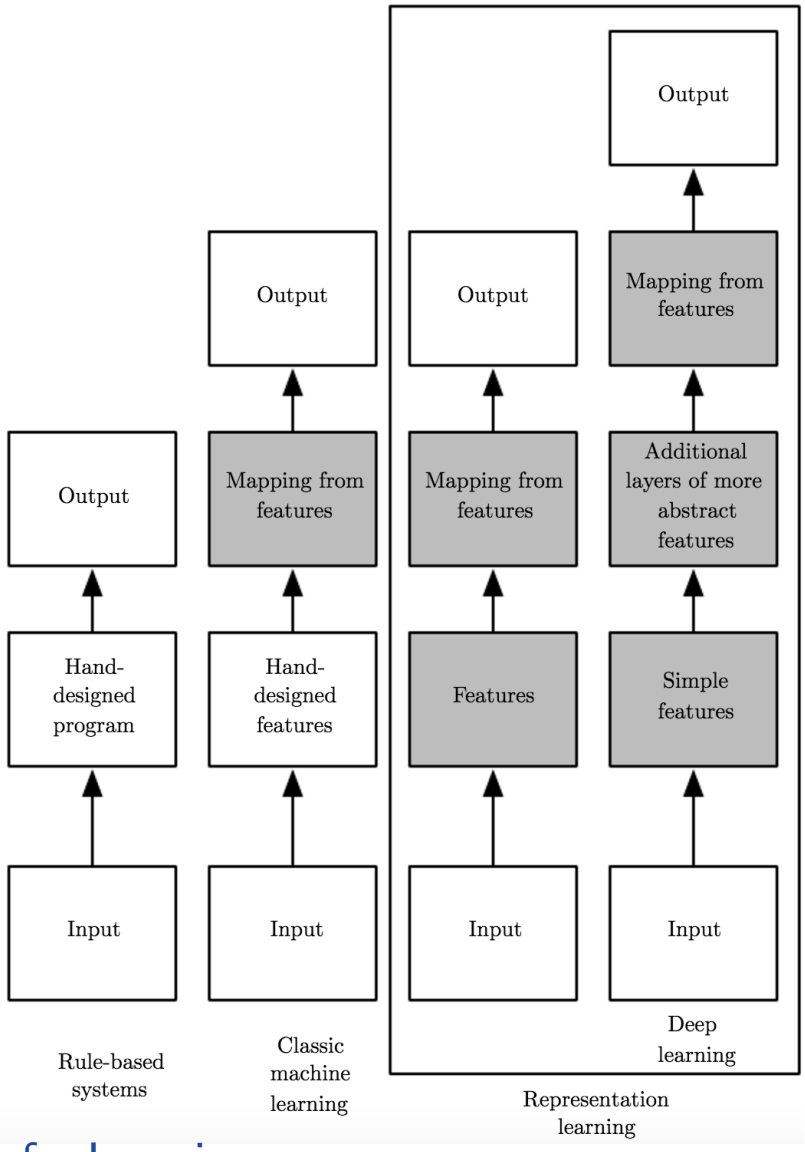


Rule-based
systems

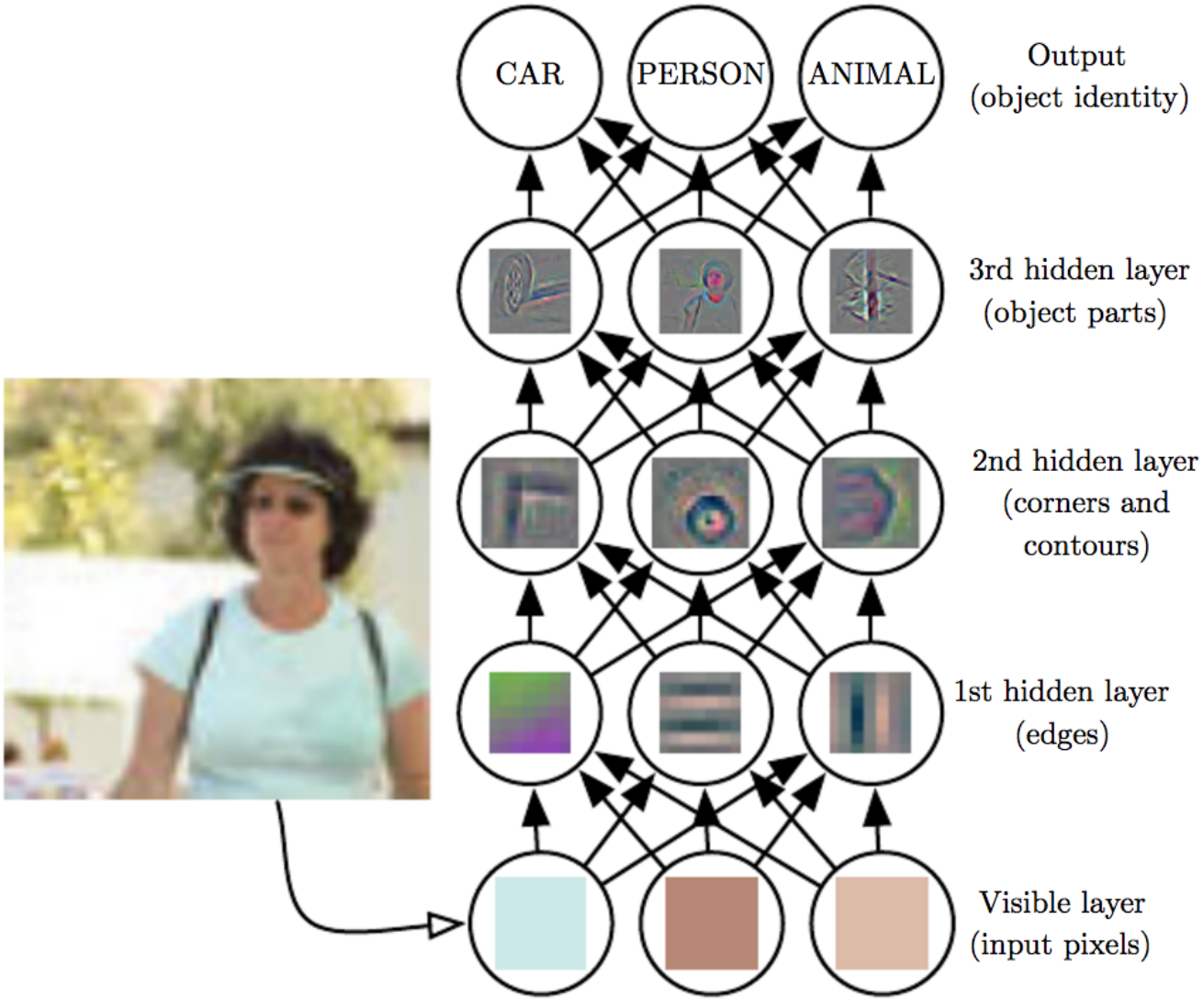
Classic Machine Learning



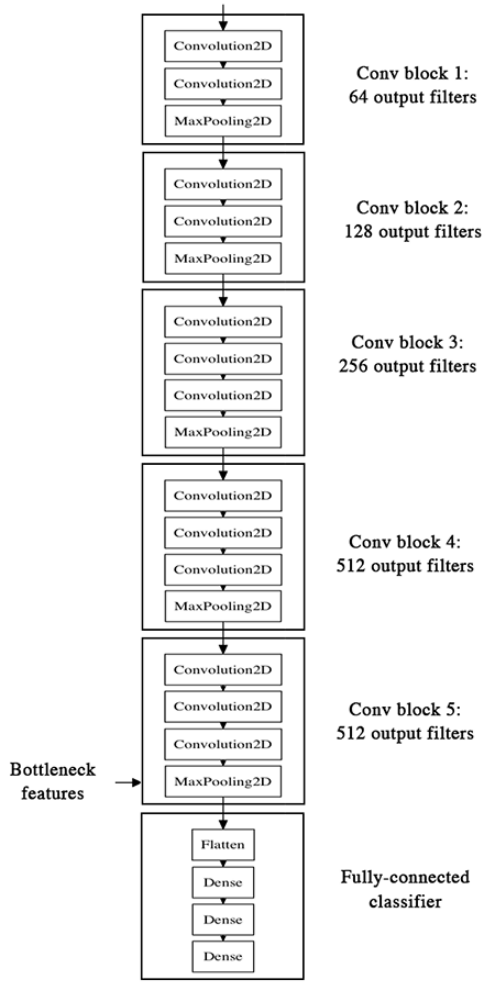
Representation Learning

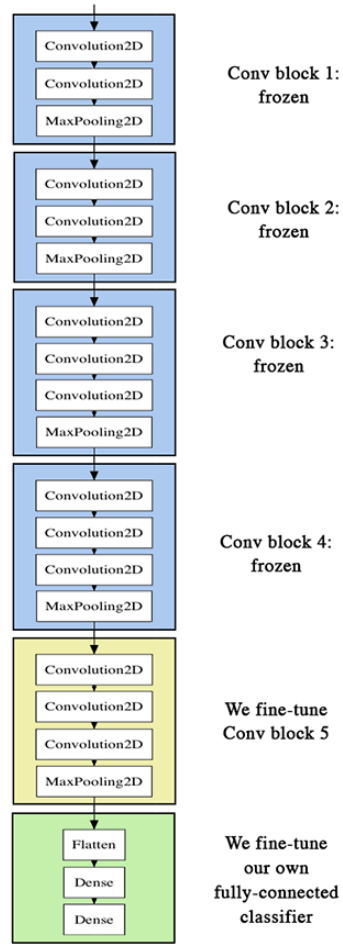


What Does a Convolutional Neural Network Learn?



Transfer Learning



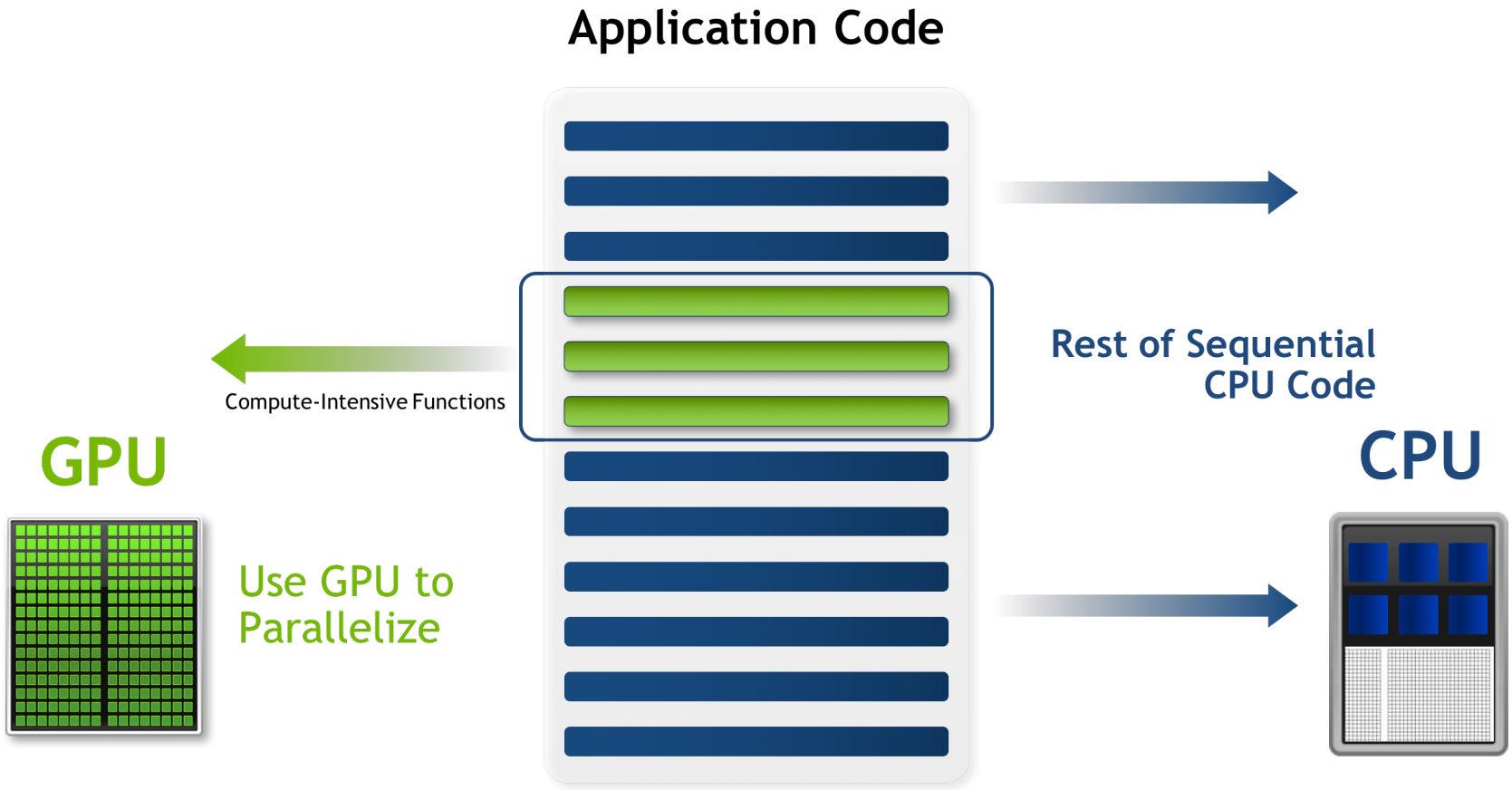


References

- **Code and Weights for Popular Deep Learning Models in Keras**: https://github.com/fchollet/deep-learning-models
- **Building powerful image classification models using very little data**: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
- https://gist.github.com/fchollet/7eb39b44eb9e16e59632d25fb3119975

Outline

GPU Acceleration



CUDA

- Historically, GPUs were used for, well, graphics processing. But, people realized that the **fine-grained parallelism** inherently in GPU architecture could be exploited for **general purpose computing**.
- CUDA (Compute Unified Device Architecture)
  - Parallel computing platform
  - Programming model and API
  - Allows enabled GPUs for general purpose processing

CUDA Ecosystem

cuBLAS

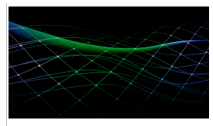
NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

MAGMA

ICL UT

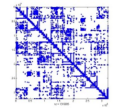
MAGMA

A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.



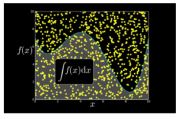
cuSOLVER

A collection of dense and sparse direct solvers which deliver significant acceleration for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications



cuSPARSE

NVIDIA CUDA Sparse (cuSPARSE) Matrix library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.



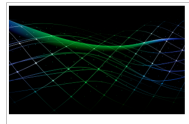
cuRAND

The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

cuDNN

cuDNN

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks, it is designed to be integrated into higher-level machine learning frameworks.



CUDA Math Library

An industry proven, highly accurate collection of standard mathematical functions, providing high performance on NVIDIA GPUs.

Thrust

Thrust

A powerful, open source library of parallel algorithms and data structures. Perform GPU-accelerated sort, scan, transform, and reductions with just a few lines of code.

cuDNN: Efficient Primitives for Deep Learning

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran
NVIDIA
Santa Clara, CA 95050
{schetlur, jwoolley, philippev, jocohen, johntran}@nvidia.com

Bryan Catanzaro
Baidu Research
Sunnyvale, CA 94089
bcatanzaro@baidu.com

Evan Shelhamer
UC Berkeley
Berkeley, CA 94720
shelhamer@eecs.berkeley.edu

cuDNN

NVIDIA® cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications:

- Convolution forward and backward, including cross-correlation
- Pooling forward and backward
- Softmax forward and backward
- Neuron activations forward and backward:
  - Rectified linear (ReLU)
  - Sigmoid
  - Hyperbolic tangent (TANH)
- Tensor transformation functions
- LRN, LCN and batch normalization forward and backward

cuDNN Accelerated Frameworks

Caffe · Chainer · DL4J Deeplearning4j · KERAS · Microsoft CNTK · MatConvNet · MINERVA · mxnet · Purine · TensorFlow · theano · torch

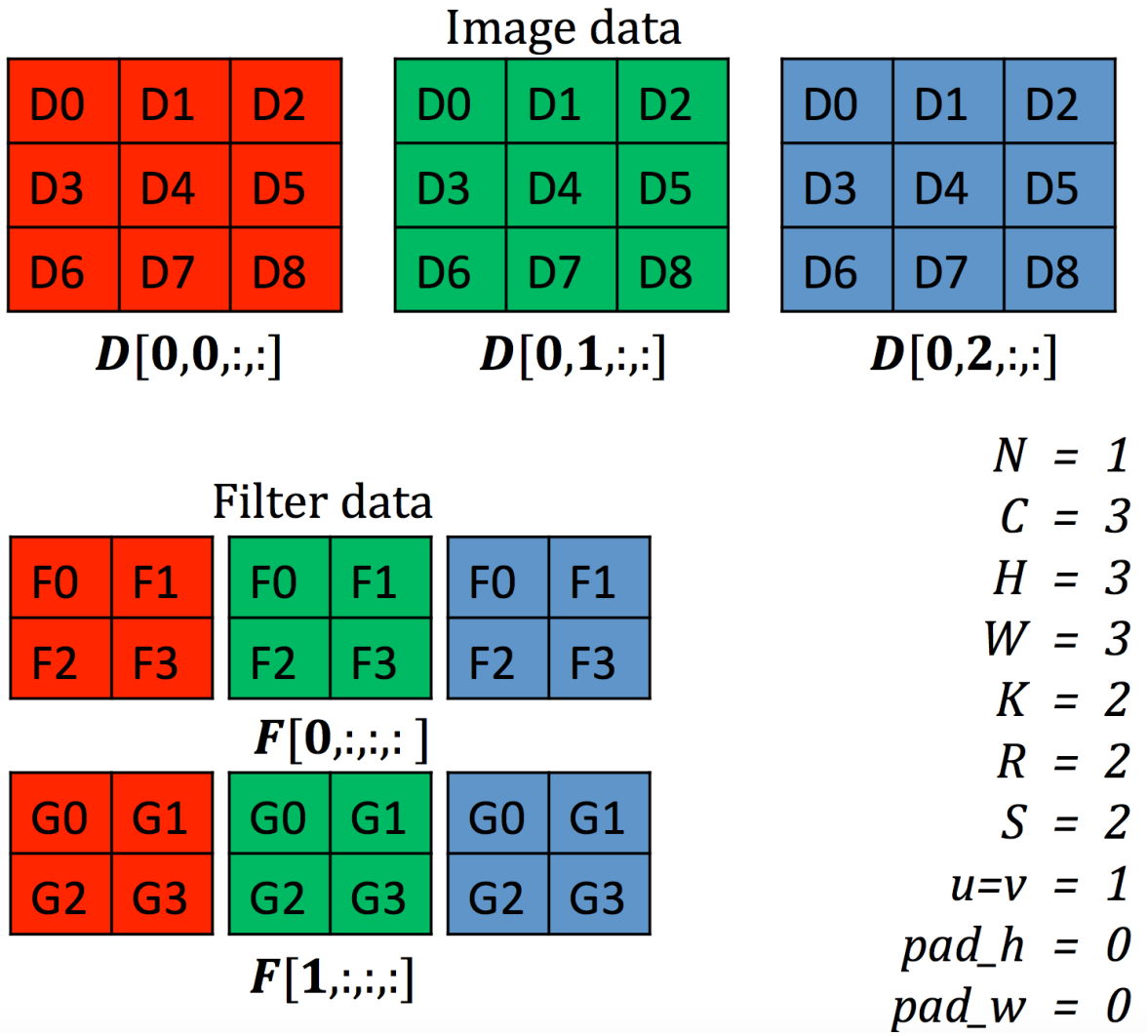
cuDNN

- 2D Convolution on Mini-Batch: $D, F \rightarrow O$
- Four dimensionals: number, depth, height, width
  - $D \in \mathbb{R}^{NCHW}$, $F \in \mathbb{R}^{KCRS}$, $O \in \mathbb{R}^{NKPQ}$

2D Convolution Mini-Batch

$$O[n,k,p,q] = O[\text{mini-batch record index}, \text{filter number}, \text{height}, \text{width}]$$
$$= \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} F[k,c,r,s] \cdot$$
$$D_0[n,c,g(p,u,R,r,pad_h),g(q,v,S,s,pad_w)]$$

2D Convolution Mini-Batch Example



2D Convolution Mini-Batch Example

$$O[n,k,p,q] = O[\text{mini-batch record index}, \text{filter number}, \text{height}, \text{width}]$$

$$= \sum_{c=0}^{2} \sum_{r=0}^{1} \sum_{s=0}^{1} F[k,c,r,s] \cdot$$

$$D_0[n,c,g(p,u=1,R=2,r,pad_h),g(q,v=1,S=2,s,pad_w=0)]$$

2D Convolution Mini-Batch Example

$$
\begin{aligned}
O[n=0,k=0,p=0,q=0] = {} & F[k=0,c=0,r=0,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=0,r=0,s=1] \cdot D_0 \ldots \\
& + F[k=0,c=0,r=1,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=0,r=1,s=1] \cdot D_0 \ldots \\
& + F[k=0,c=1,r=0,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=1,r=0,s=1] \cdot D_0 \ldots \\
& + F[k=0,c=1,r=1,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=1,r=1,s=1] \cdot D_0 \ldots \\
& + F[k=0,c=2,r=0,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=2,r=0,s=1] \cdot D_0 \ldots \\
& + F[k=0,c=2,r=1,s=0] \cdot D_0 \ldots \\
& + F[k=0,c=2,r=1,s=1] \cdot D_0 \ldots
\end{aligned}
$$

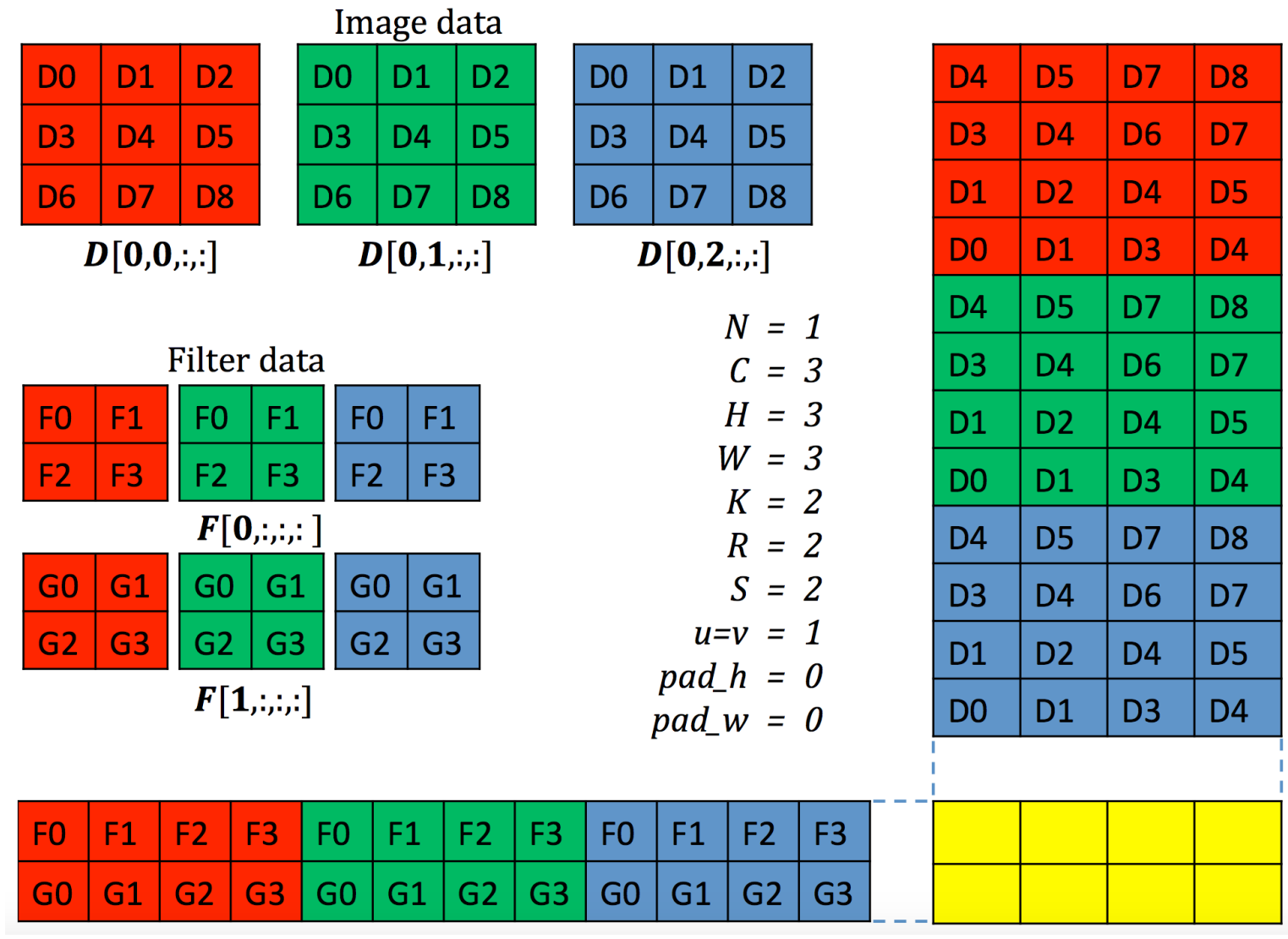
2D Convolution Mini-Batch Example

$$
\begin{aligned}
O[n=0,k=0,p=0,q=0] = & F0_0 \cdot D0_0 \\
& + F1_0 \cdot D1_0 \\
& + F2_0 \cdot D3_0 \\
& + F3_0 \cdot D4_0 \\
& + F0_1 \cdot D0_1 \\
& + F1_1 \cdot D1_1 \\
& + F2_1 \cdot D3_1 \\
& + F3_1 \cdot D4_1 \\
& + F0_2 \cdot D0_2 \\
& + F1_2 \cdot D1_2 \\
& + F2_2 \cdot D3_2 \\
& + F3_2 \cdot D4_2
\end{aligned}
$$

2D Convolution Mini-Batch Example

$$
\begin{aligned}
O[n=0,k=0,p=0,q=1] = & F0_0 \cdot D1_0 \\
& + F1_0 \cdot D2_0 \\
& + F2_0 \cdot D4_0 \\
& + F3_0 \cdot D5_0 \\
& + F0_1 \cdot D1_1 \\
& + F1_1 \cdot D2_1 \\
& + F2_1 \cdot D4_1 \\
& + F3_1 \cdot D5_1 \\
& + F0_2 \cdot D1_2 \\
& + F1_2 \cdot D2_2 \\
& + F2_2 \cdot D4_2 \\
& + F3_2 \cdot D5_2
\end{aligned}
$$

2D Convolution Mini-Batch Lowering Example



GPU Acceleration on AlexNet

GPU ACCELERATION

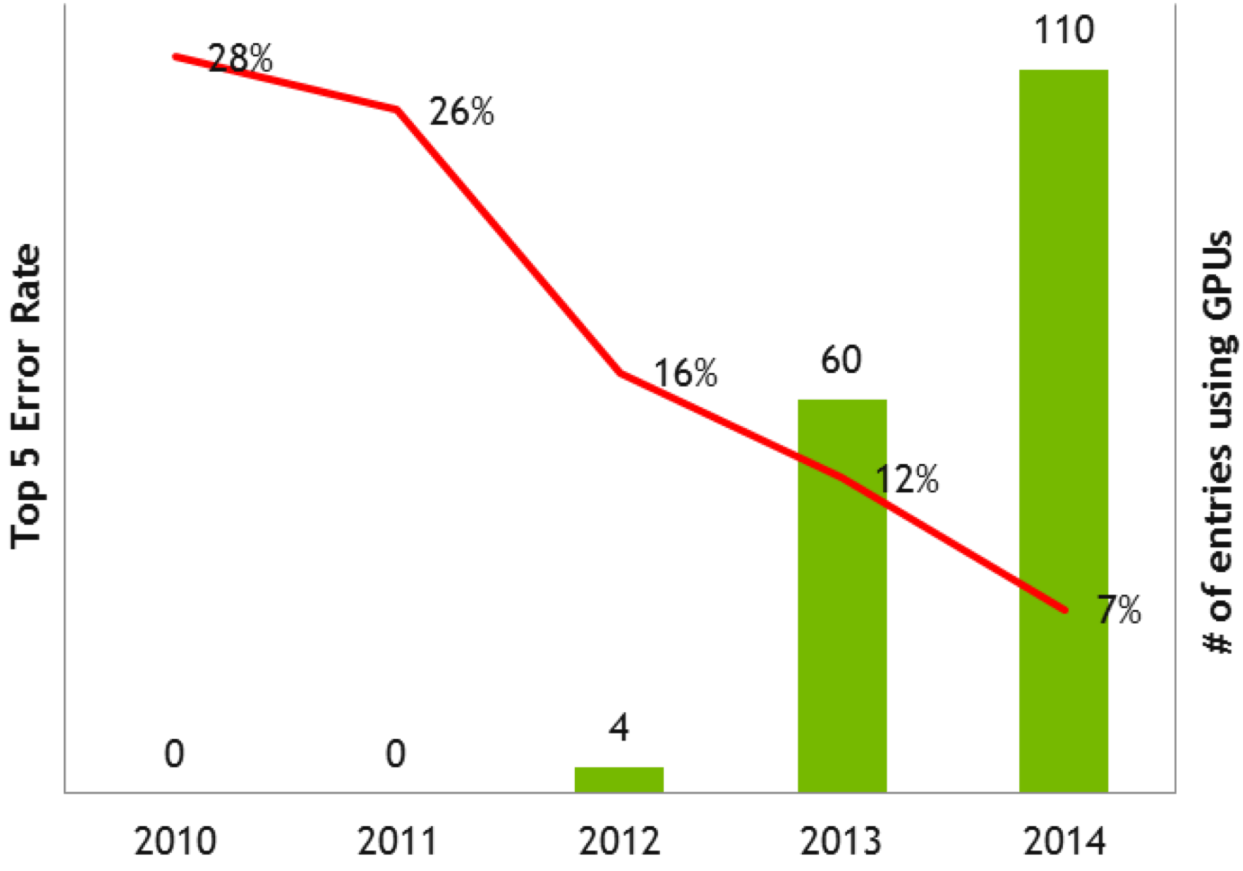
Training A Deep, Convolutional Neural Network

Batch Size	Training Time CPU	Training Time GPU	GPU Speed Up
64 images	64 s	7.5 s	8.5X
128 images	124 s	14.5 s	8.5X
256 images	257 s	28.5 s	9.0X

- ILSVRC12 winning model: “Supervision”
- 7 layers
- 5 convolutional layers + 2 fully-connected
- ReLU, pooling, drop-out, response normalization
- Implemented with Caffe

- Dual 10-core Ivy Bridge CPUs
- 1 Tesla K40 GPU
- CPU times utilized Intel MKL BLAS library
- GPU acceleration from CUDA matrix libraries (cuBLAS)

ILSVRC Entries Using GPUs



Outline

References

- **Awesome - Most Cited Deep Learning Papers**: https://github.com/terryum/awesome-deep-learning-papers
- **Discover the current state of the art in objects classification.**: http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

Deep Learning People to Follow

- Geoff Hinton (Univ. of Toronto)
- Yoshua Bengio (Univ. of Montreal)
- Yann LeCun (NYU, Facebook)
- Ian Goodfellow (Google Brain, Generative Adversarial Networks)
- Ilya Sutskever (OpenAI, AlexNet)
- Matthew Zeiler (NYU, ZFNet)
- Rob Fergus (NYU, Facebook, ZFNet)
- Christian Szegedy, Sergey Ioffe (Google, GoogLeNet, Batch Normalization)
- Andrej Karpathy (OpenAI)
- Kaiming He (Facebook, Residual Networks)
- Francois Chollet (Google, Keras)