TRANSFER LEARNING

TRAINING HUGE MODELS IS A BOTTLENECK

Requires managing huge dataset

Takes a long time → more difficult to tune hyperparameters

Expensive

- Many hours on rented GPU instance(s)
- Electricity cost on owned hardware

But we want to use powerful models for our own problems

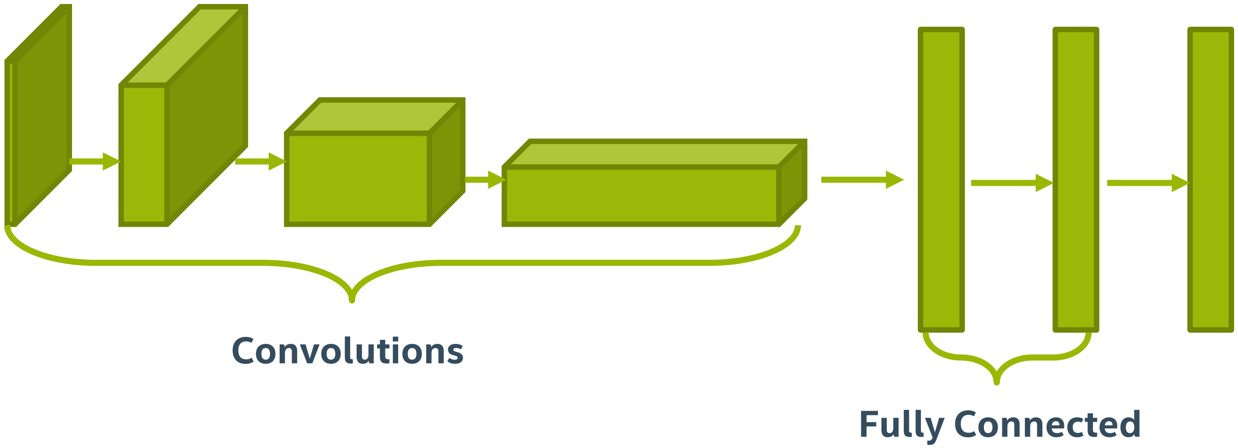
TRANSFER LEARNING

Idea: layers in trained model might generalize

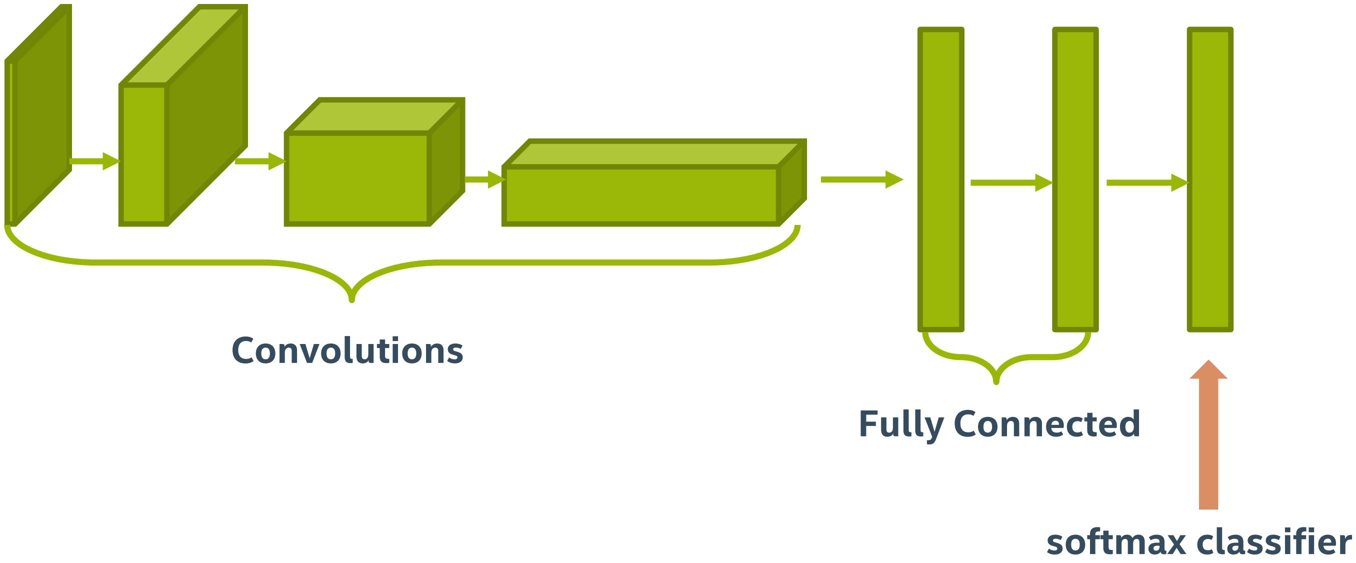
If we just change the later layers, can use previously trained model to solve new problem!

Nice thing: many pre-trained models available

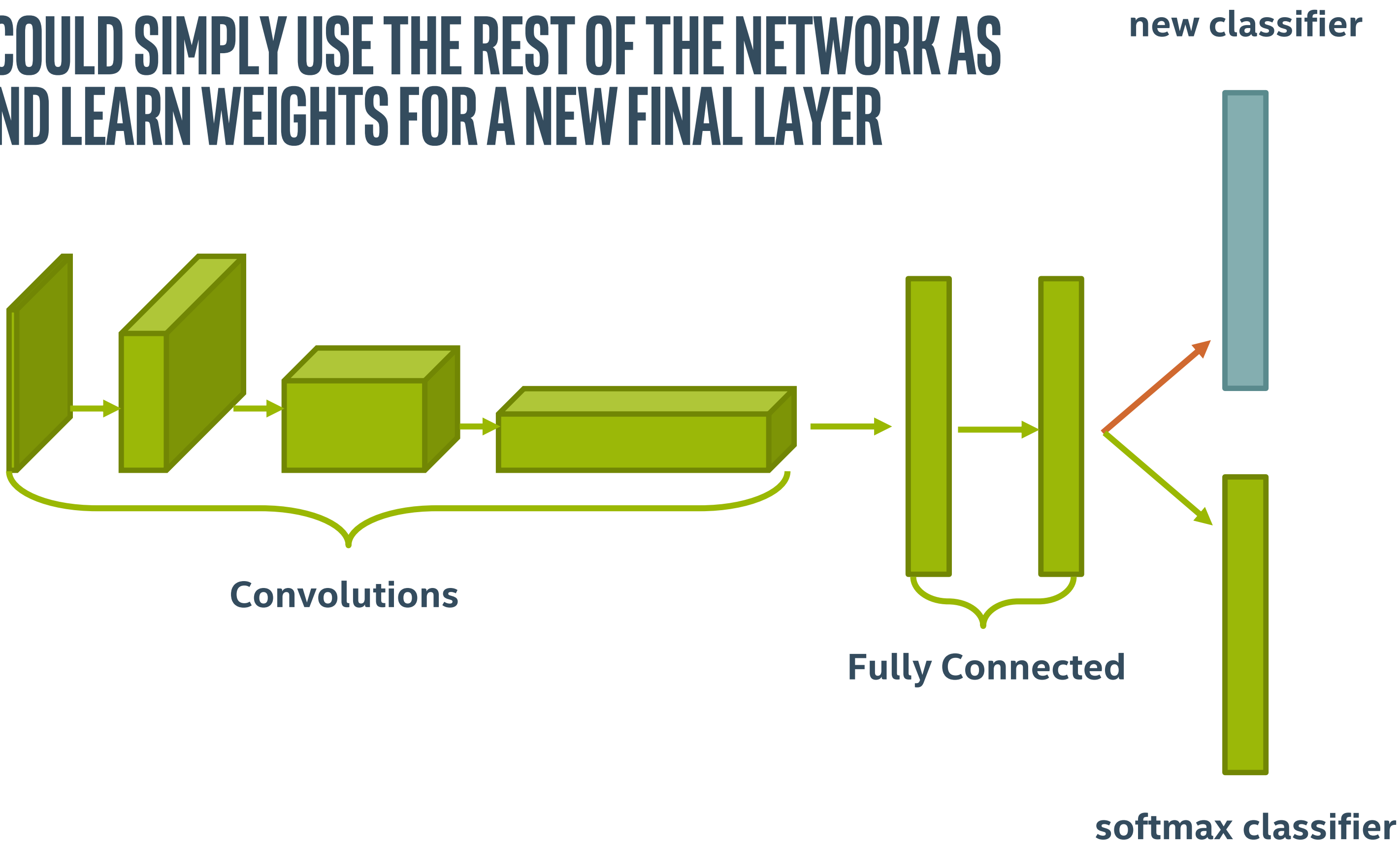
OUR GENERAL CNN TEMPLATE



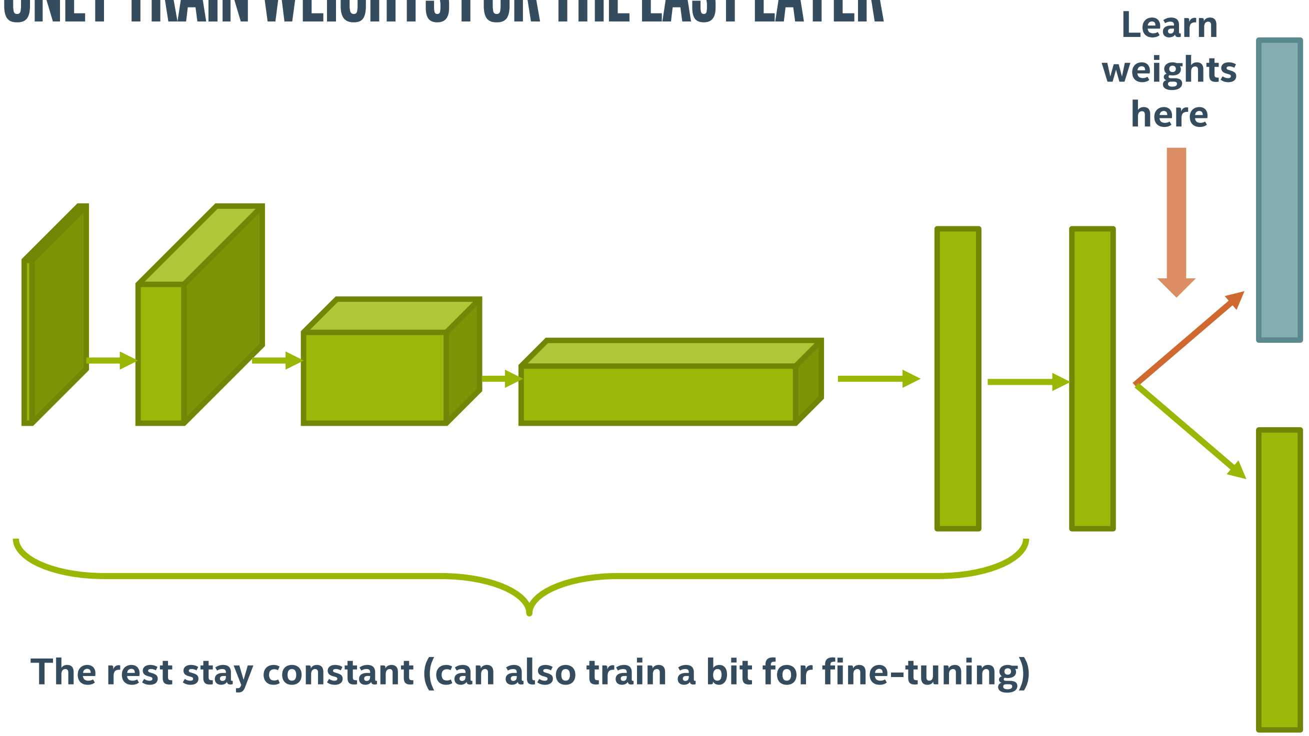
WHAT IF THE MOST DECISION-MAKING FOR CLASSIFICATION COMES AT FINAL LAYER?



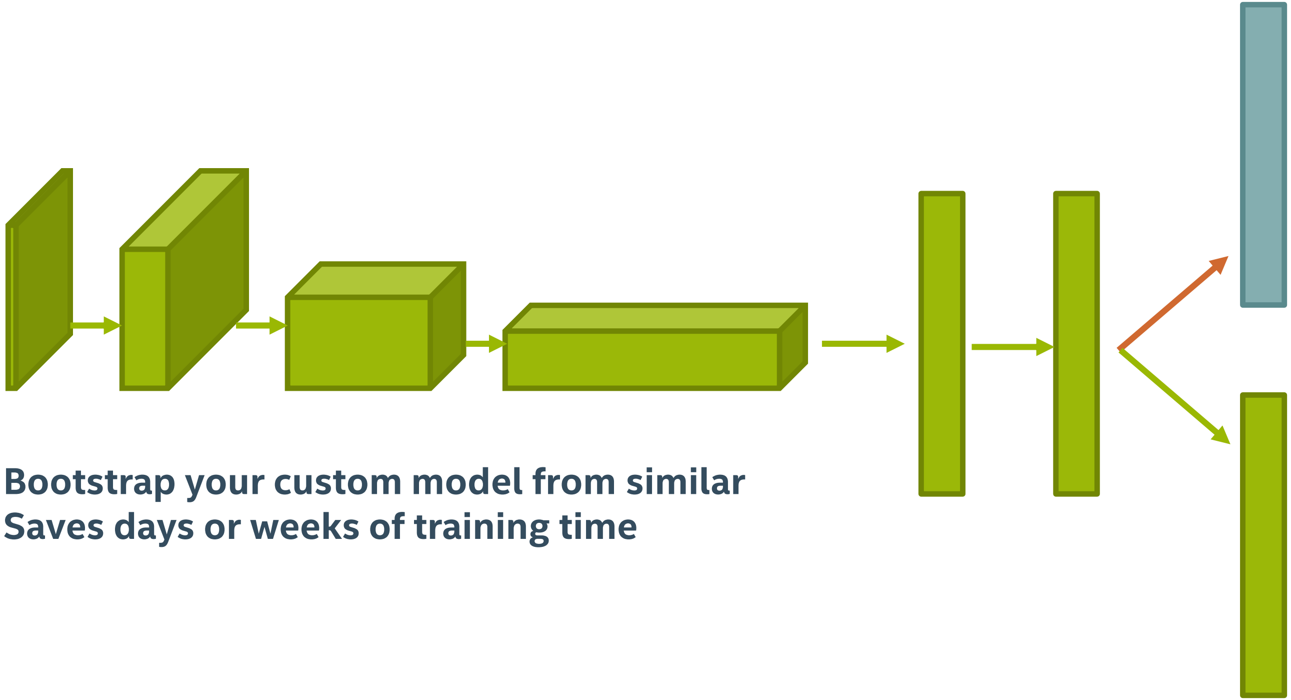
WE COULD SIMPLY USE THE REST OF THE NETWORK AS IS AND LEARN WEIGHTS FOR A NEW FINAL LAYER



WE ONLY TRAIN WEIGHTS FOR THE LAST LAYER



THIS TECHNIQUE IS CALLED TRANSFER LEARNING



Bootstrap your custom model from similar
Saves days or weeks of training time

TRANSFER LEARNING IN TENSORFLOW (CLASSIFICATION)

1. **Get handle to output from second-to-last layer**
2. **Create a new fully connected layer**
   - Number of neurons equal to the number of output classes
3. **Create new softmax cross-entropy loss**
4. **Create a training op to minimize the new loss**
   - Set var_list parameter to be just the new layer variables
5. **Train with new data!**

ACCESSING OPS AND TENSORS

ACCESSING ARBITRARY OPERATIONS/TENSORS

In general, we have been simply holding onto pointers for our Tensor objects in TensorFlow:

```
c = tf.multiply(a, b)  # c is a Tensor object
```

But what if we don't have pointer due to using pre-built models or layer functions which don't return handles?

ACCESSING ARBITRARY OPERATIONS/TENSORS

`graph.get_operation_by_name()`

`graph.get_tensor_by_name()`

functions allow us to get handles to Operations/Tensors by passing in a string name:

```
with graph.as_default():
  tf.multiply(a, b, name='mul')  # forgot to assign handle!
  c = graph.get_tensor_by_name('mul:0')  # got Tensor handle!
```

OP NAMES

Be careful when getting handles to Operations vs Tensors

Operation names are the names you pass in as name argument (they don't have number at end)

```
c_op = graph.get_operation_by_name('mul')
```

TENSOR NAMES

Tensors have numeric suffix along with parent Op's name

Number corresponds to the output index from Op

```
c = graph.get_tensor_by_name('mul:0')
```

Some Ops have multiple output Tensors

- example: tf.nn.moments()

```
mean = graph.get_tensor_by_name('moments:0')
var  = graph.get_tensor_by_name('moments:1')
```

BATCH NORMALIZATION

Ioffe and Szegedy

INTERNAL COVARIATE SHIFT

Change in distributions of activations during training due to weights changing

Slows down training

- Weights have to keep figuring out how to respond to different activations

If we can keep neuron activations (more) normally distributed, weights can settle down more quickly

BATCH NORMALIZATION

Idea: normalize ("whiten") the activations of each layer

$$\hat{a}^{(l)} = \frac{a^{(l)} - \bar{a}^{(l)}}{\sqrt{Var(a^{(l)}) + \epsilon}}$$

⟵ **Epsilon ϵ prevents divide by zero**

Then add learnable "weight" and "bias" terms for final output

$$BN\left(a^{(l)}\right) = \gamma \hat{a}^{(l)} + \beta$$

$BN\left(a^{(l)}\right)$ **is what gets passed to next layer**

Note: paper uses x and y to refer to initial activations and final activations.
Changed here for consistency with class

BATCH NORM: TRAINING VS FINAL

During training, get mean ($\bar{a}^{(l)}$) and variance ($Var(a^{(l)})$) of activations w.r.t. *training batch* (not single example)

After training, get mean and variance of activations over the *entire training set*.

Alternative: keep moving average of mean/variance during training.

IMPLEMENTATION NOTE

With batch normalization, we no longer need our standard bias term, b

$z^{(l)} = a^{(l-1)}W^{(l-1)} + b$ or $z^{(l)} = conv(a^{(l-1)}W^{(l-1)} + b)$

Due to the normalization subtracting the mean, the bias term gets cancelled out

Simply need weights $W^{(l-1)}$

$z^{(l)} = a^{(l-1)}W^{(l-1)}$ or $z^{(l)} = conv(a^{(l-1)}W^{(l-1)})$

BATCH NORMALIZATION IN TENSORFLOW

Method 1: Manually

```
z = tf.matmul(a_prev, W)
a = tf.nn.relu(z)

a_mean, a_var = tf.nn.moments(a, [0])

scale = tf.Variable(tf.ones([depth/channels]))
beta = tf.Variable(tf.zeros ([depth/channels]))

bn = tf.nn.batch_normalizaton(a, a_mean, a_var, beta, scale, 1e-3)
```

For testing, replace a_mean, a_var with full training statistics

BATCH NORMALIZATION IN TENSORFLOW

Method 2: Built-in layer function

```
z = tf.matmul(a_prev, W)
a = tf.nn.relu(z)

bn = tf.layers.batch_normalization(a)
```

- Keeps a decaying average of activation mean and variance
- Much simpler than doing manually
- Can still replace with full training mean/variance if desired

RESULTS OF BATCH NORMALIZATION

Speeds up training dramatically

Enables increased learning rate with less chance of exploding gradients

Reduces effectiveness/need for dropout

Final model is more accurate

In summary: use batch normalization.

DEEPER NETWORKS AND RECEPTIVE FIELD

VGGNet

VGG NET

Very Deep Convolutional Networks for Large-Scale Image Recognition

- Karen Simonyan and Andrew Zisserman, 2014

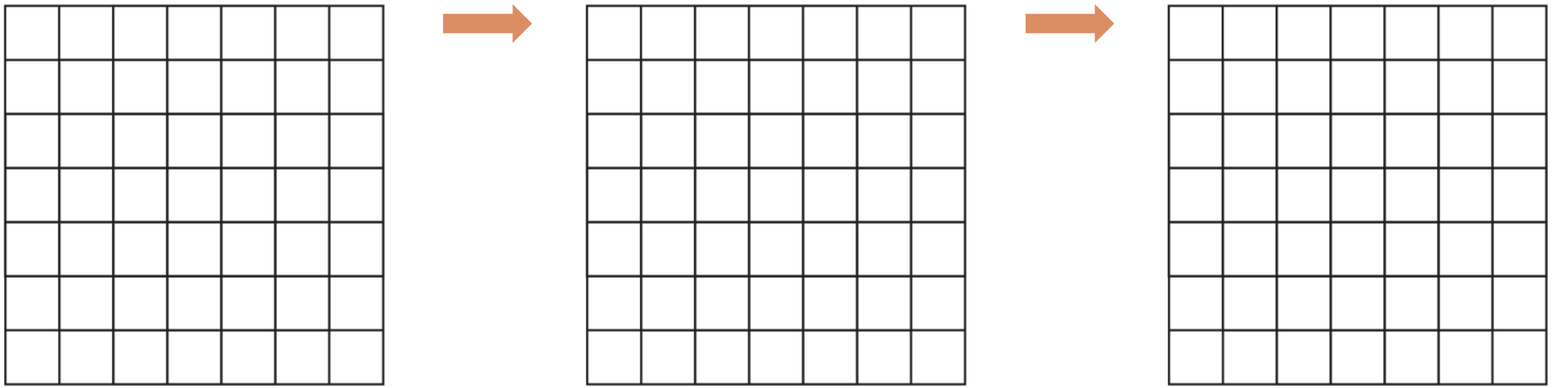
Problem: networks involve many manual decisions

How to choose between different size convolutions?

Idea: Simplify network and add many more layers

RECEPTIVE FIELD

Idea: Each subsequent 3x3 convolution effectively "sees" a larger portion of the inputs



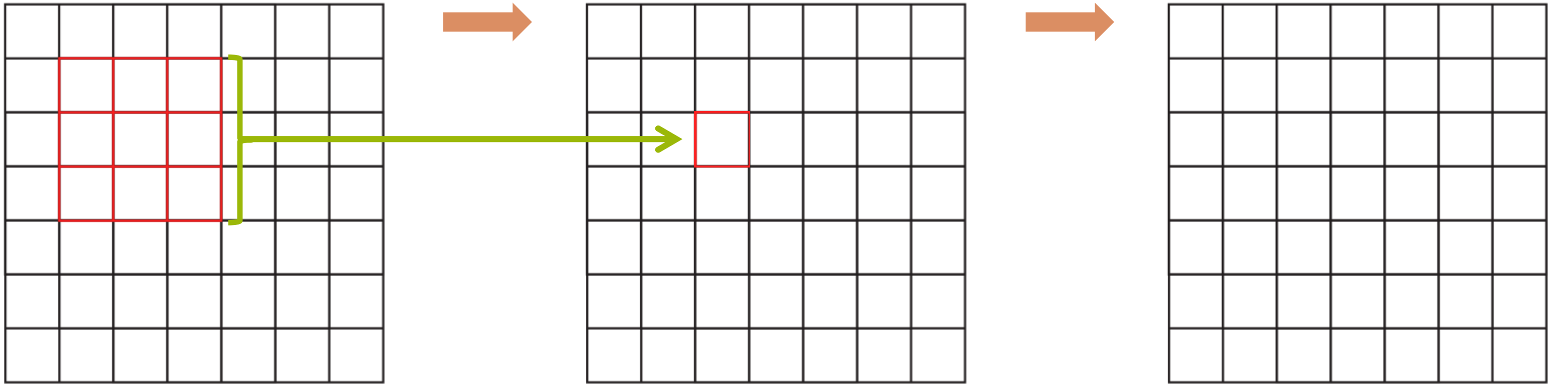
Layer 1
(Input)

Layer 2

Layer 3

RECEPTIVE FIELD

A single output in Layer 2 is the result of “seeing” a 3x3 grid from Layer 1



Layer 1

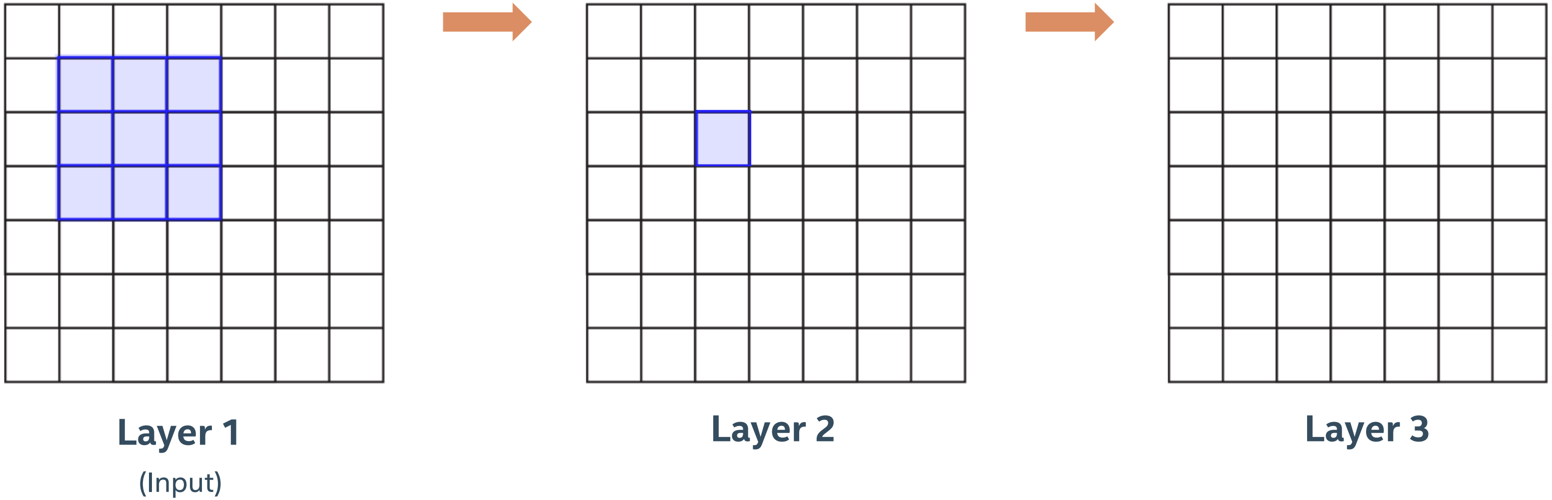
(Input)

Layer 2

Layer 3

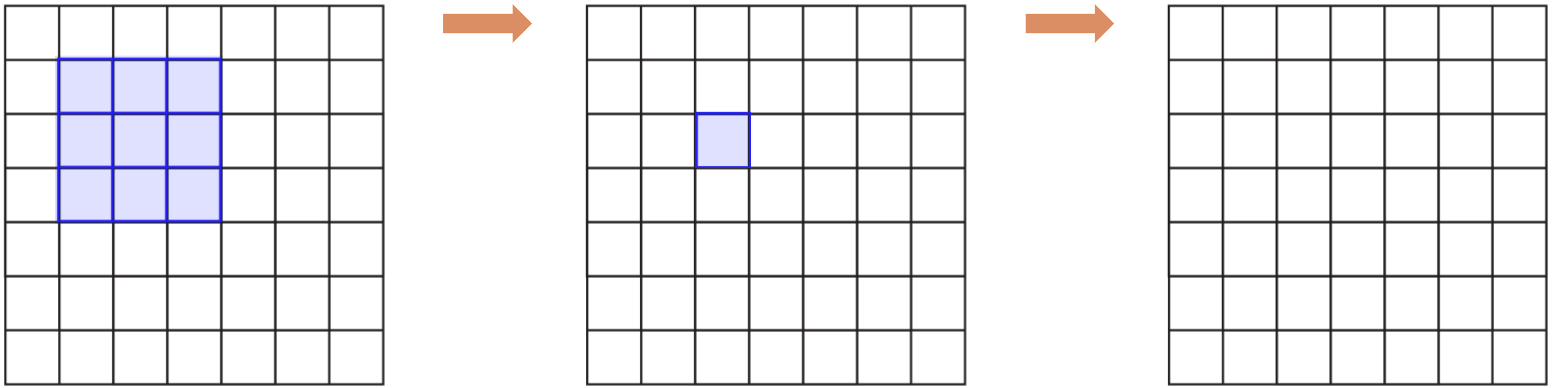
RECEPTIVE FIELD

We can say that the "receptive field" of layer 2 is 3x3.
Each output has been influenced by a 3x3 patch of inputs.



RECEPTIVE FIELD

What about on layer 3?



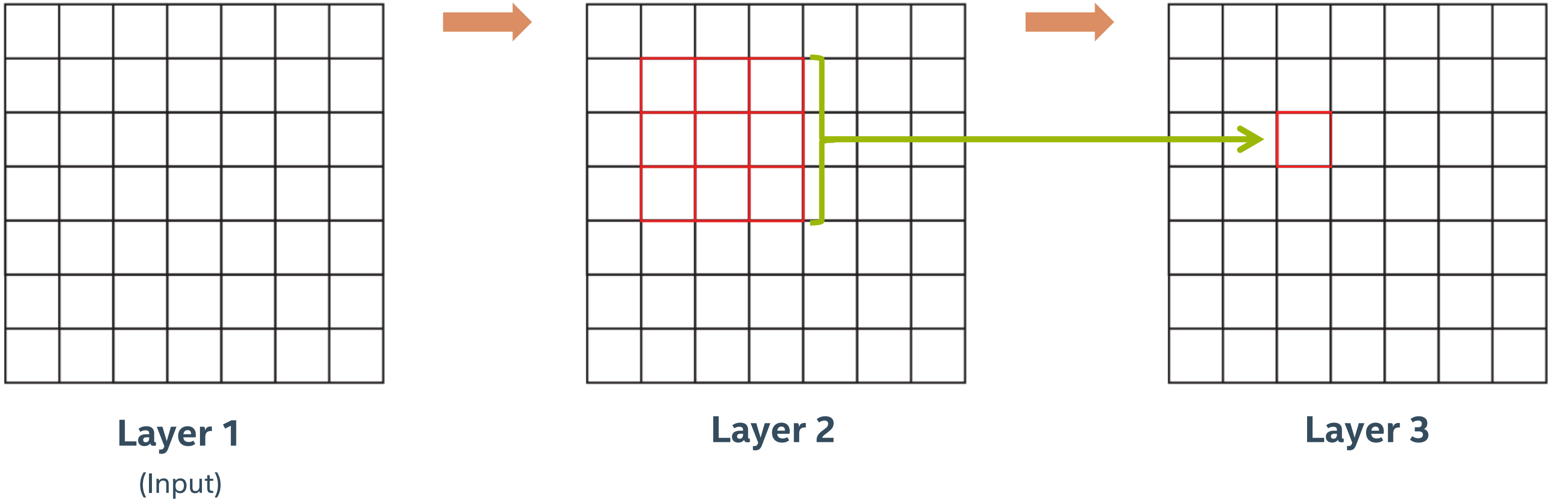
Layer 1
(Input)

Layer 2

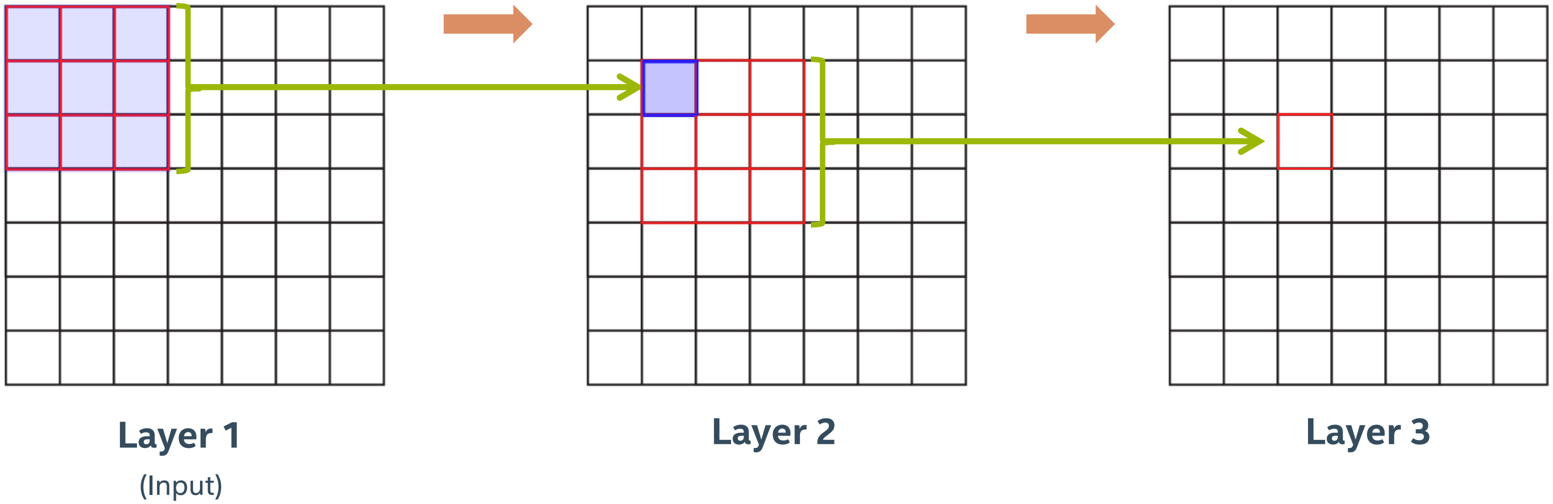
Layer 3

RECEPTIVE FIELD

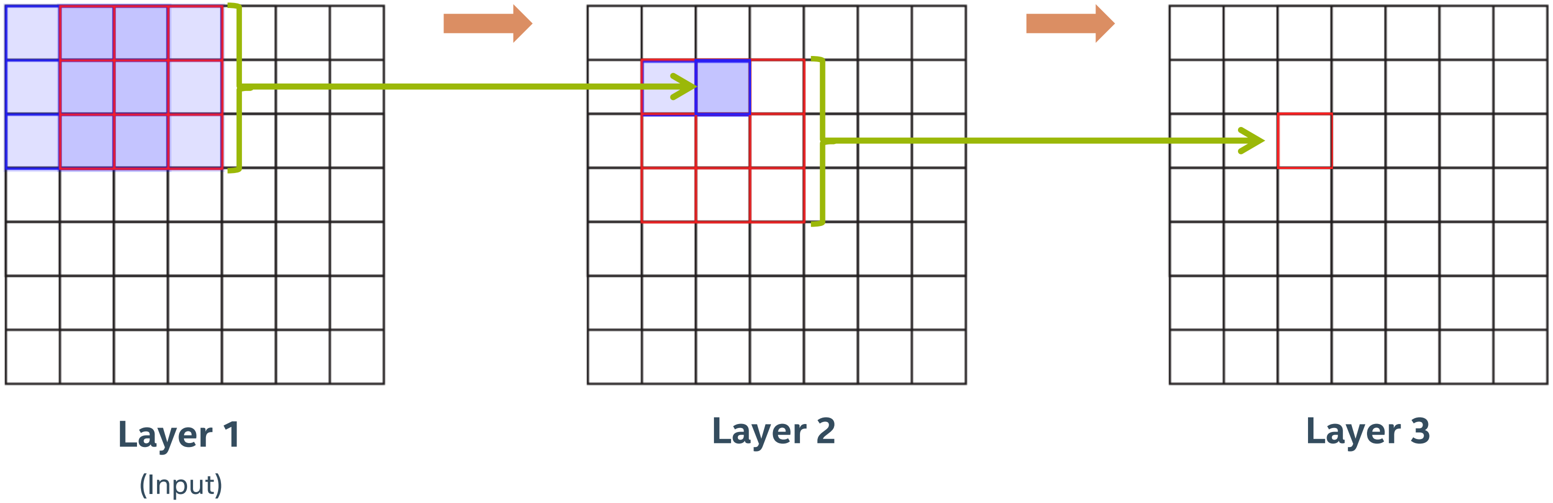
This output on Layer 3 uses a 3x3 patch from layer 2.
How much from layer 1 does it use?



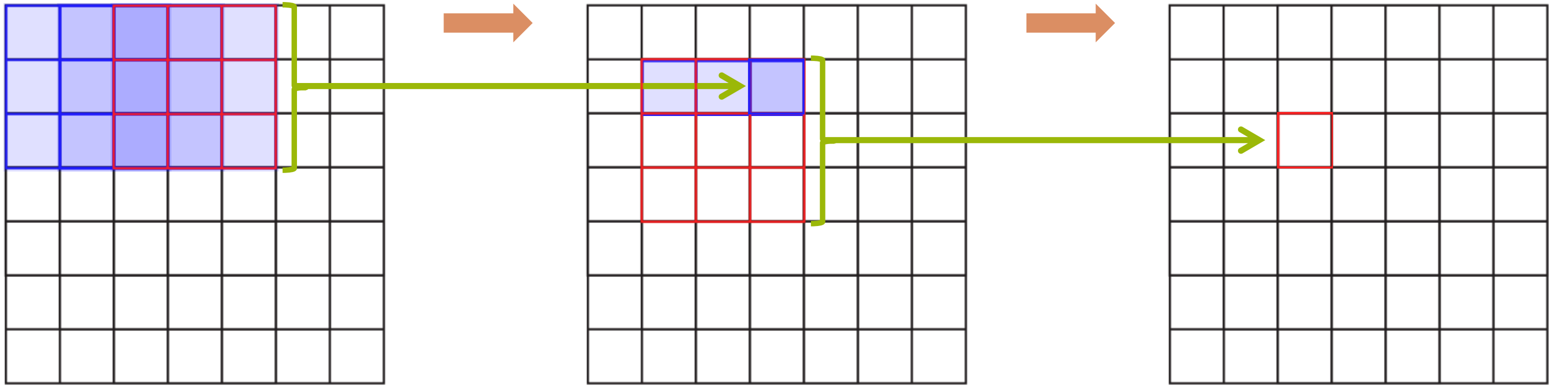
RECEPTIVE FIELD



RECEPTIVE FIELD



RECEPTIVE FIELD



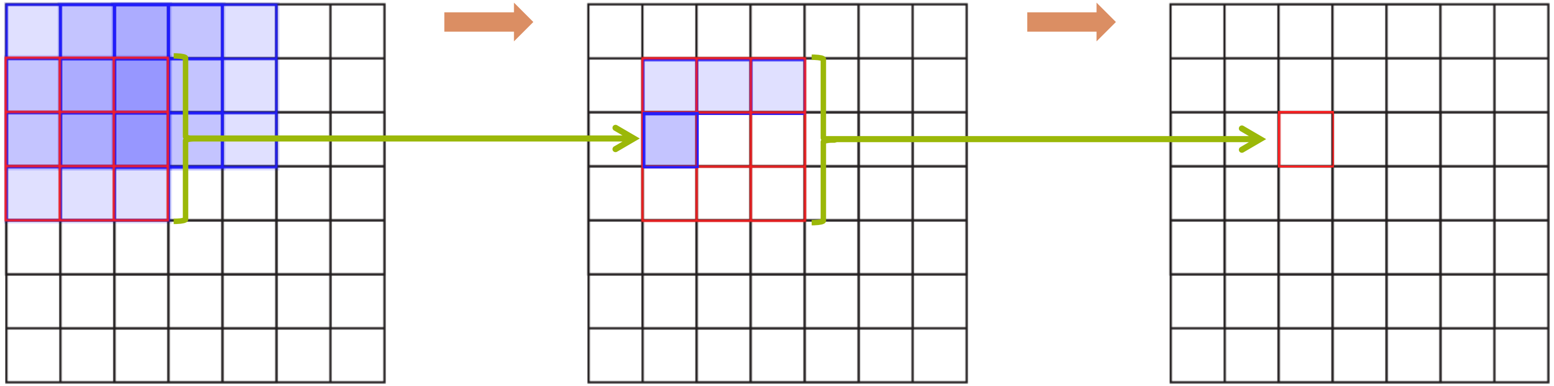
Layer 1

(Input)

Layer 2

Layer 3

RECEPTIVE FIELD



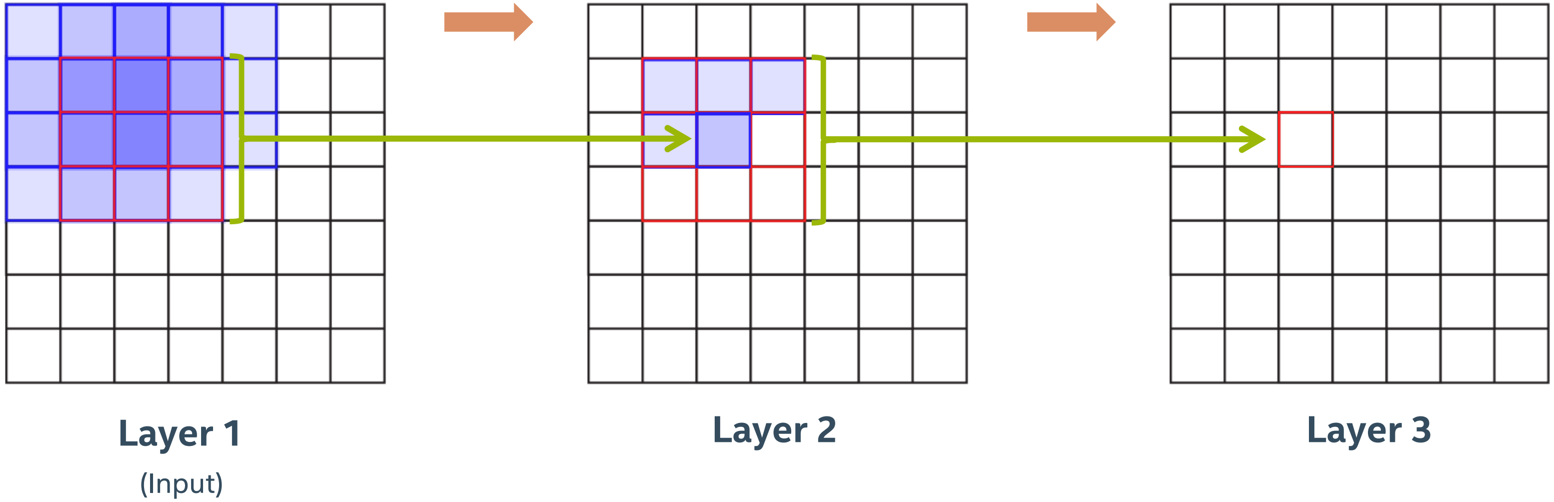
Layer 1

(Input)

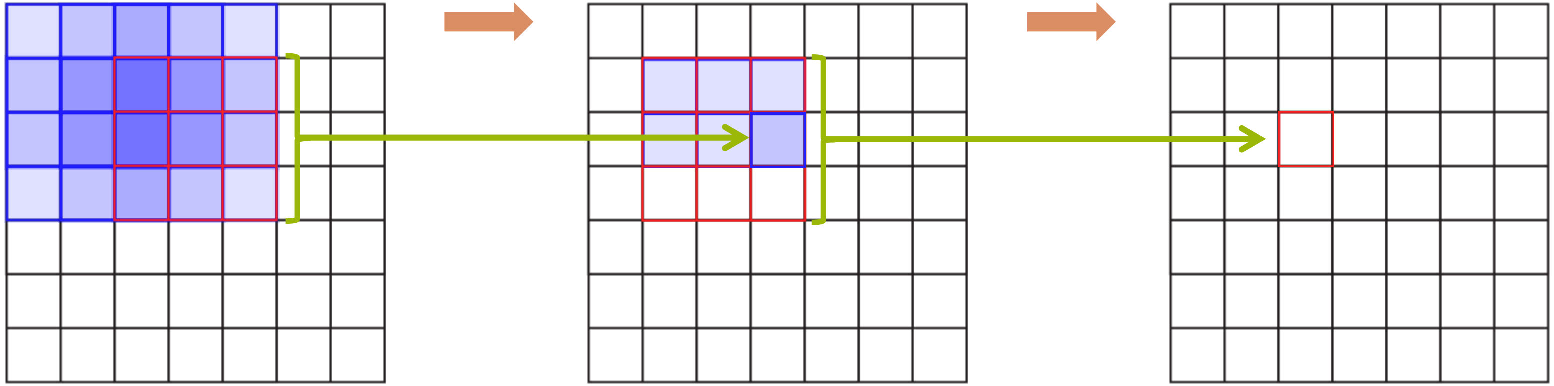
Layer 2

Layer 3

RECEPTIVE FIELD



RECEPTIVE FIELD



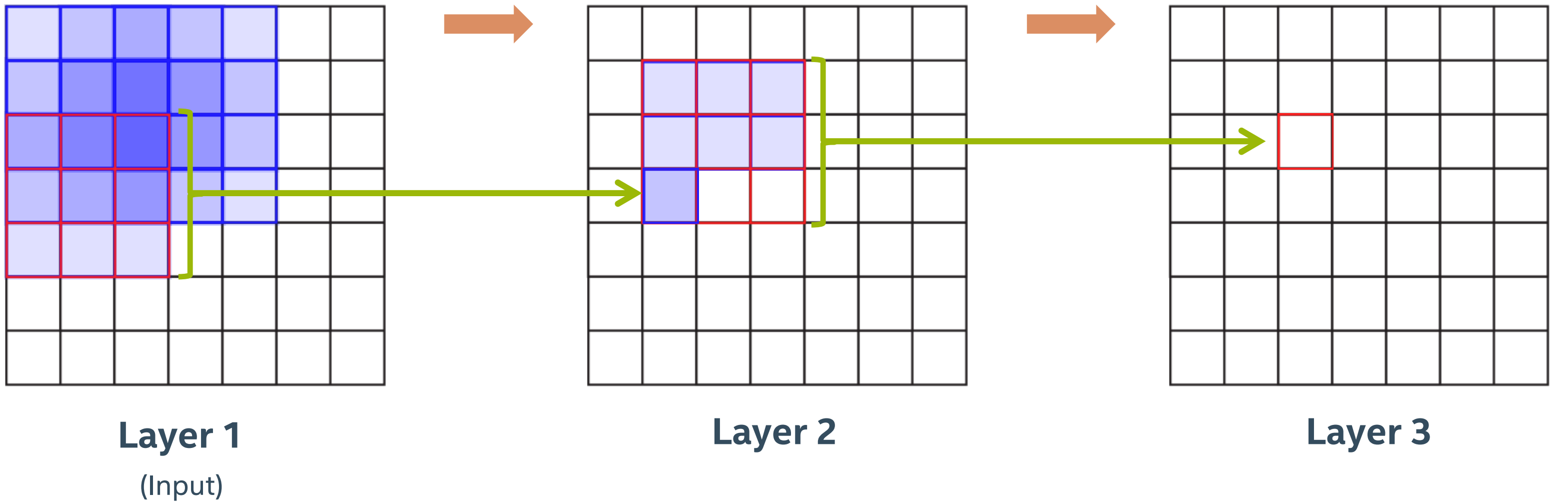
Layer 1

(Input)

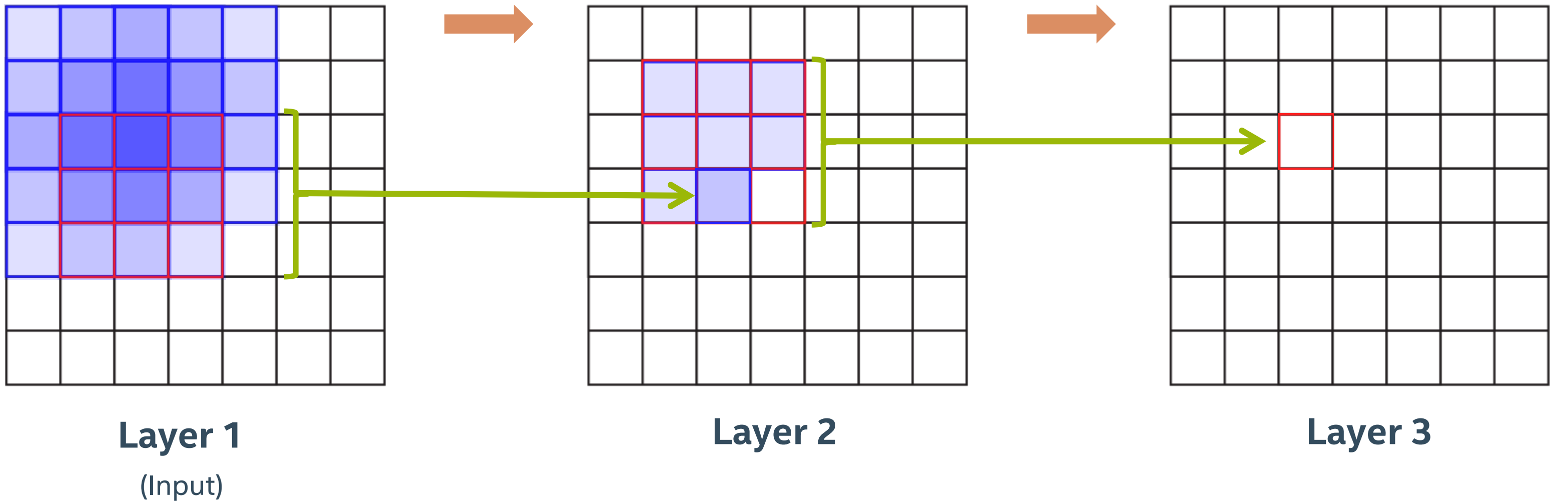
Layer 2

Layer 3

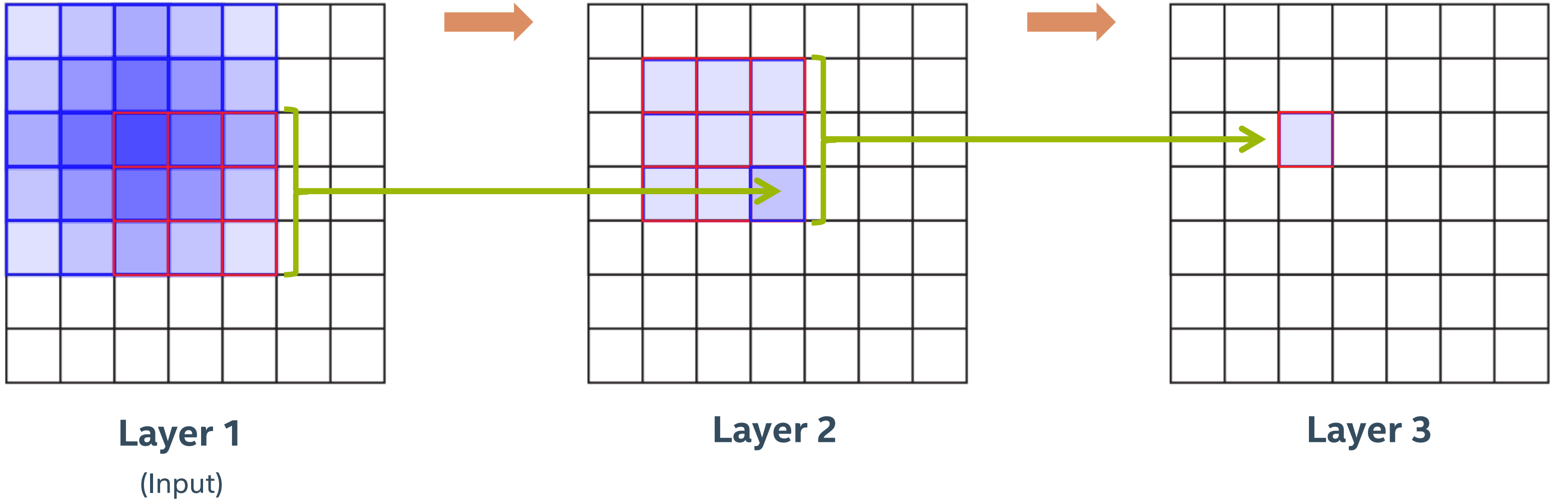
RECEPTIVE FIELD



RECEPTIVE FIELD

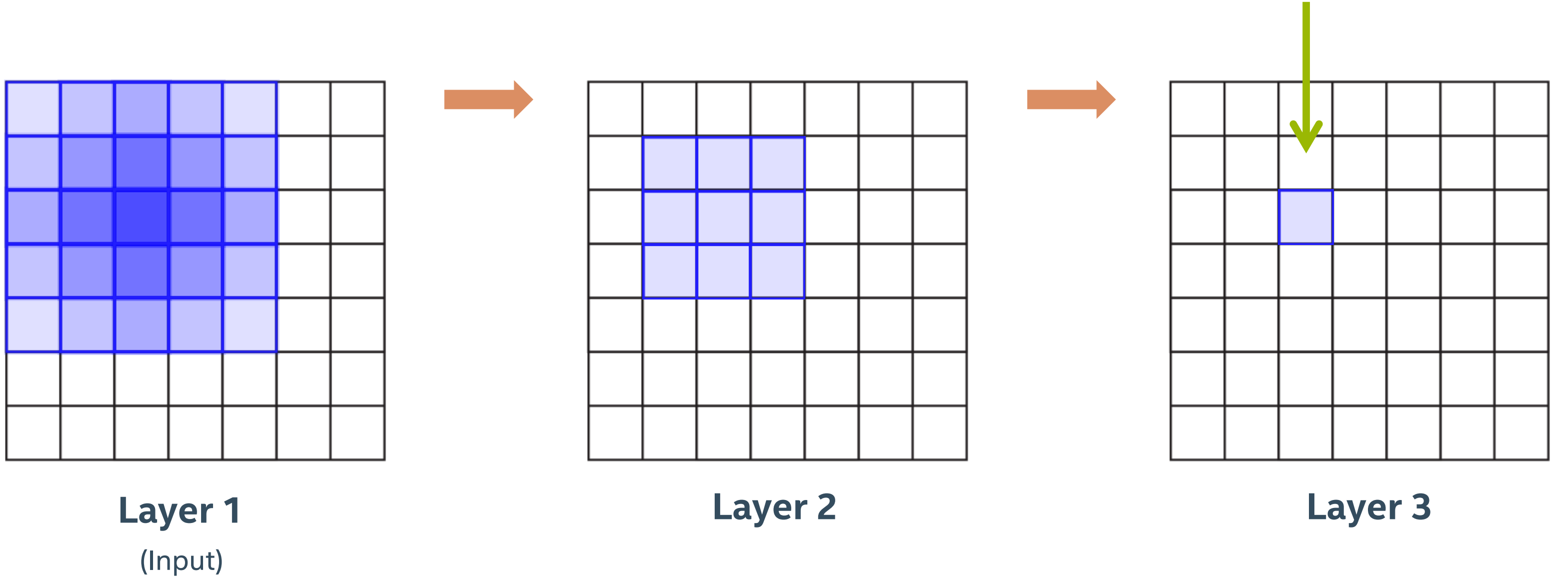


RECEPTIVE FIELD



RECEPTIVE FIELD

Each square in layer 3 "sees" a 5x5 grid from layer 1.



RECEPTIVE FIELD

Two 3x3, stride 1 convolutions in a row is effectively a 5x5

Three 3x3 convolutions is effectively a 7x7 convolution

Benefit: fewer parameters!

One 3x3 layer

$3 \times 3 \times C \times C = 9C^2$

Three 3x3 layers

$3 \times (9C^2) = 27C^2$

assume C input/output channels

One 7x7 layer

$7 \times 7 \times C \times C = 49C^2$

$49C^2 \rightarrow 27C^2 \rightarrow \approx$45% reduction!

RAMIFICATIONS OF VGGNET

One of the first papers to experiment with many layers

- More is better!

Can use multiple 3x3 convolutions to simulate larger kernels with fewer parameters

Served as "base model" for future works

Nervana™ AI Academy