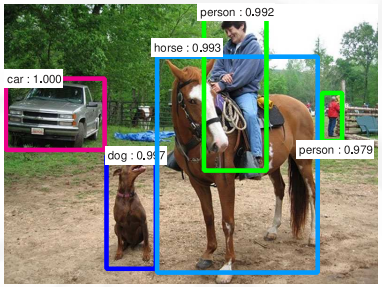
NPFL114, Lecture 07

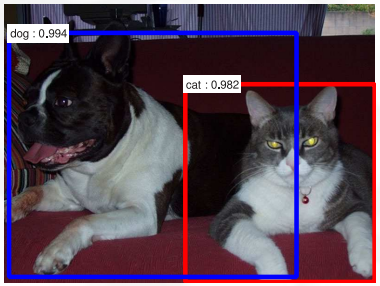
Object Detection & Segmentation, Recurrent Neural Networks

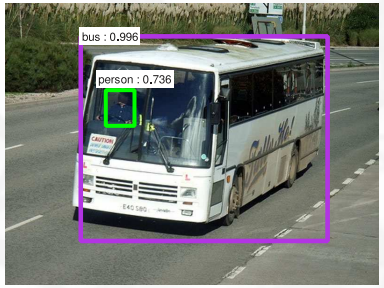
Milan Straka

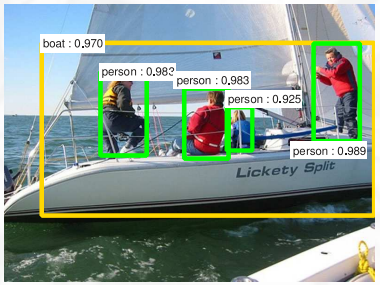
Object Detection

- Object detection (including location)



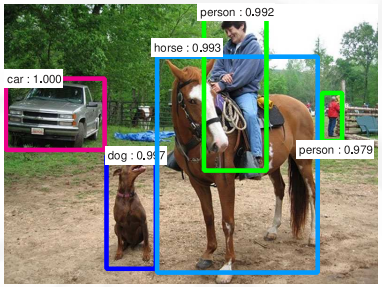


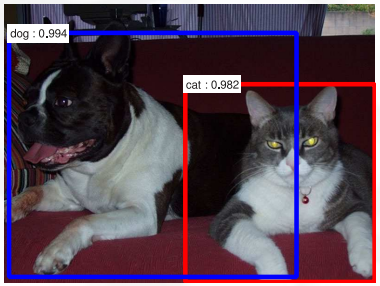


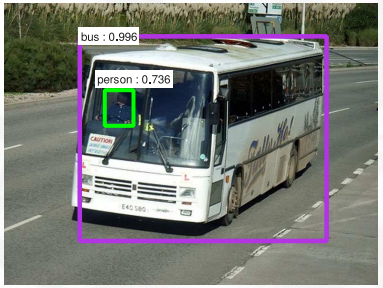


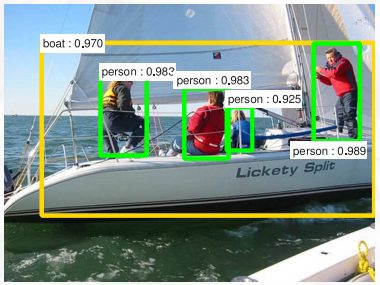
Object Detection

- Object detection (including location)

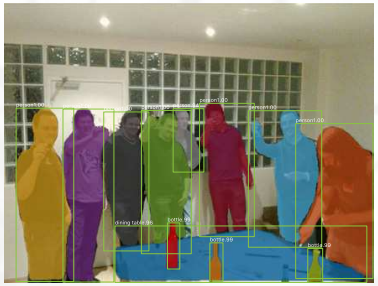
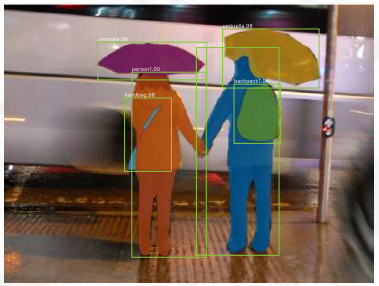
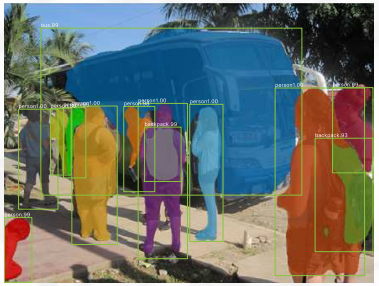
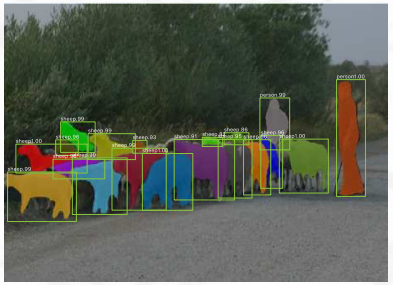






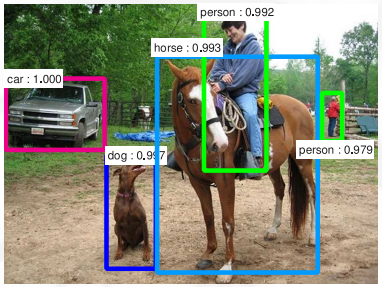


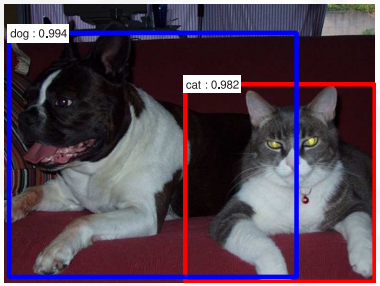
- Image segmentation

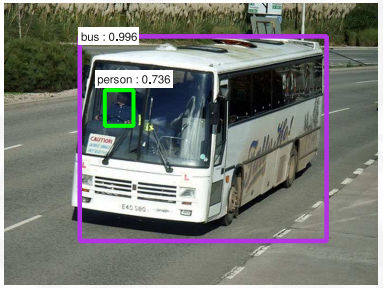





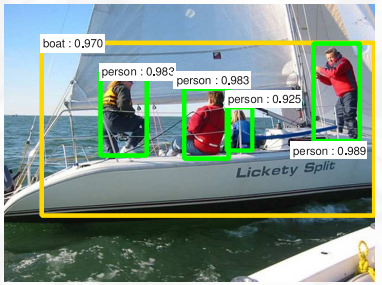
Object Detection

- Object detection (including location)

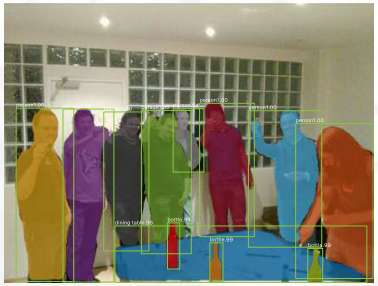
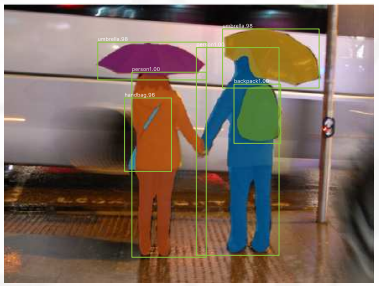
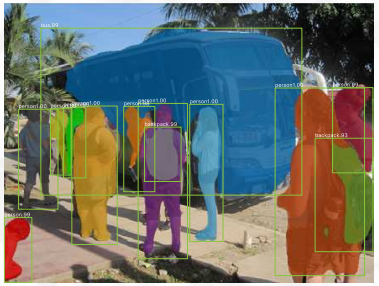
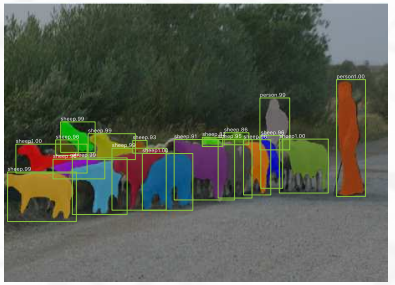




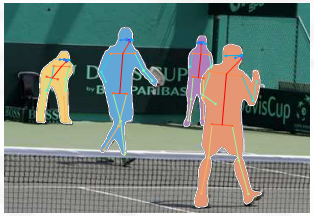
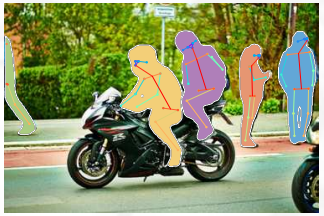




- Image segmentation

- Human pose estimation


Fast R-CNN

- Start with a network pre-trained on ImageNet (VGG-16 is used in the original paper).

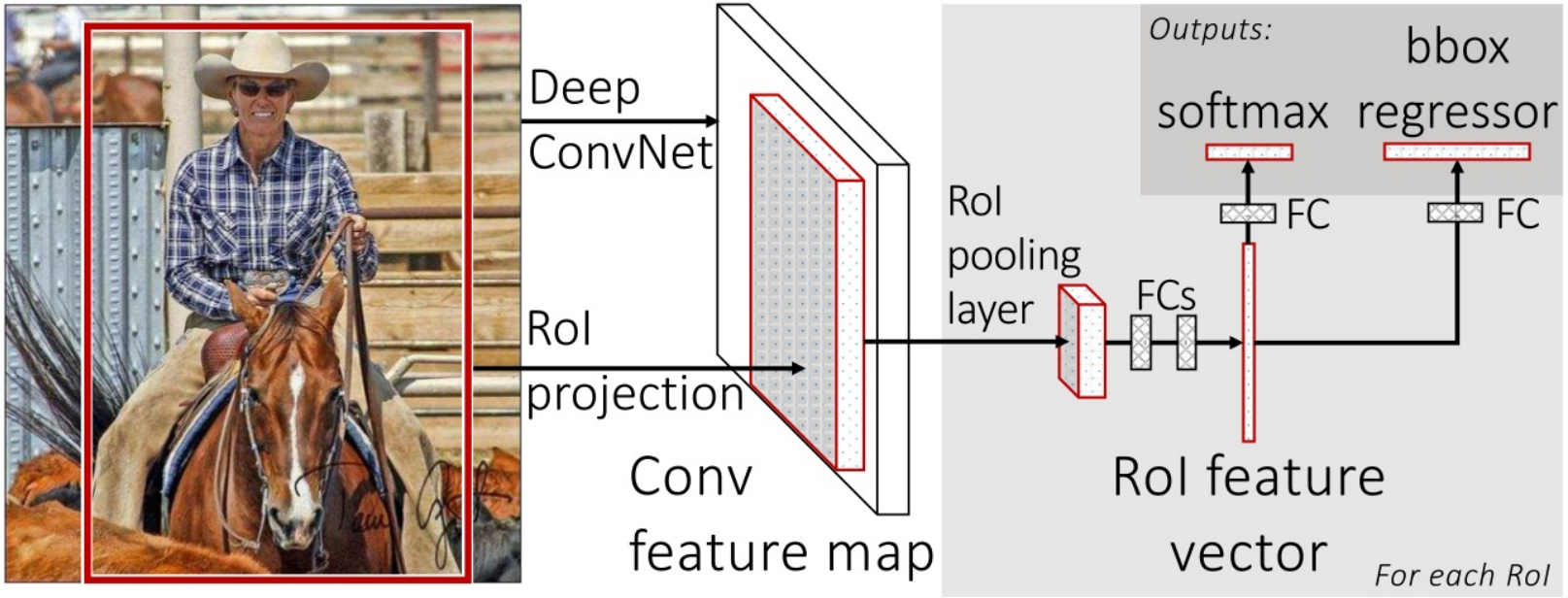
Fast R-CNN

- Start with a network pre-trained on ImageNet (VGG-16 is used in the original paper).

RoI Pooling

- Crucial for fast performance.
- The last max-pool layer ($14 \times 14 \rightarrow 7 \times 7$ in VGG) is replaced by a RoI pooling layer, producing output of the same size. For each output sub-window we max-pool the corresponding values in the output layer.
- Two sibling layers are added, one predicting $K + 1$ categories and the other one predicting 4 bounding box parameters for each of K categories.

Fast R-CNN


Deep
ConvNet
RoI
projection
Conv
feature map
RoI
pooling
layer
FCs
Outputs:
softmax
bbox
regressor
FC
FC
RoI feature
vector
For each RoI

Fast R-CNN

The bounding box is parametrized as follows. Let x_r, y_r, w_r, h_r be center coordinates and width and height of the RoI, and let x, y, w, h be parameters of the bounding box. We represent them as follows:

$$t_x = (x - x_r)/w_r, \quad t_y = (y - y_r)/h_r$$
$$t_w = \log(w/w_r), \quad t_h = \log(h/h_r)$$

Fast R-CNN

The bounding box is parametrized as follows. Let x_r, y_r, w_r, h_r be center coordinates and width and height of the RoI, and let x, y, w, h be parameters of the bounding box. We represent them as follows:

$$t_x = (x - x_r)/w_r, \quad t_y = (y - y_r)/h_r$$
$$t_w = \log(w/w_r), \quad t_h = \log(h/h_r)$$

Usually a smooth_{L_1} loss is employed for bounding box parameters:

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

Fast R-CNN

The bounding box is parametrized as follows. Let x_r, y_r, w_r, h_r be center coordinates and width and height of the RoI, and let x, y, w, h be parameters of the bounding box. We represent them as follows:

$$t_x = (x - x_r)/w_r, \quad t_y = (y - y_r)/h_r$$
$$t_w = \log(w/w_r), \quad t_h = \log(h/h_r)$$

Usually a smooth_{L_1} loss is employed for bounding box parameters:

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

The complete loss is then

$$L(\hat{c}, \hat{t}, c, t) = L_{\text{cls}}(\hat{c}, c) + \lambda[c \geq 1] \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}(\hat{t}_i - t_i).$$

Intersection over union

For two bounding boxes (or two masks) the *intersection over union* (*IoU*) is a ration of the intersection of the boxes (or masks) and the union of the boxes (or masks).

Intersection over union

For two bounding boxes (or two masks) the *intersection over union* (*IoU*) is a ration of the intersection of the boxes (or masks) and the union of the boxes (or masks).

Choosing RoIs for training

During training, we use 2 images with 64 RoIs each. The RoIs are selected so that 25% have intersection over union (IoU) overlap with ground-truth boxes at least 0.5; the others are chosen to have the IoU in range $[0.1, 0.5)$.

Intersection over union

For two bounding boxes (or two masks) the *intersection over union* (*IoU*) is a ration of the intersection of the boxes (or masks) and the union of the boxes (or masks).

Choosing RoIs for training

During training, we use 2 images with 64 RoIs each. The RoIs are selected so that 25% have intersection over union (IoU) overlap with ground-truth boxes at least 0.5; the others are chosen to have the IoU in range $[0.1, 0.5)$.

Choosing RoIs during inference

Single object can be found in multiple RoIs. To choose the most salient one, we perform *non-maximum suppression* — we ignore RoIs which have an overlap with a higher scoring larger than a given threshold (usually, 0.3 is used).

Object Detection Evaluation

Average Precision

Evaluation is performed using *Average Precision* (*AP*).

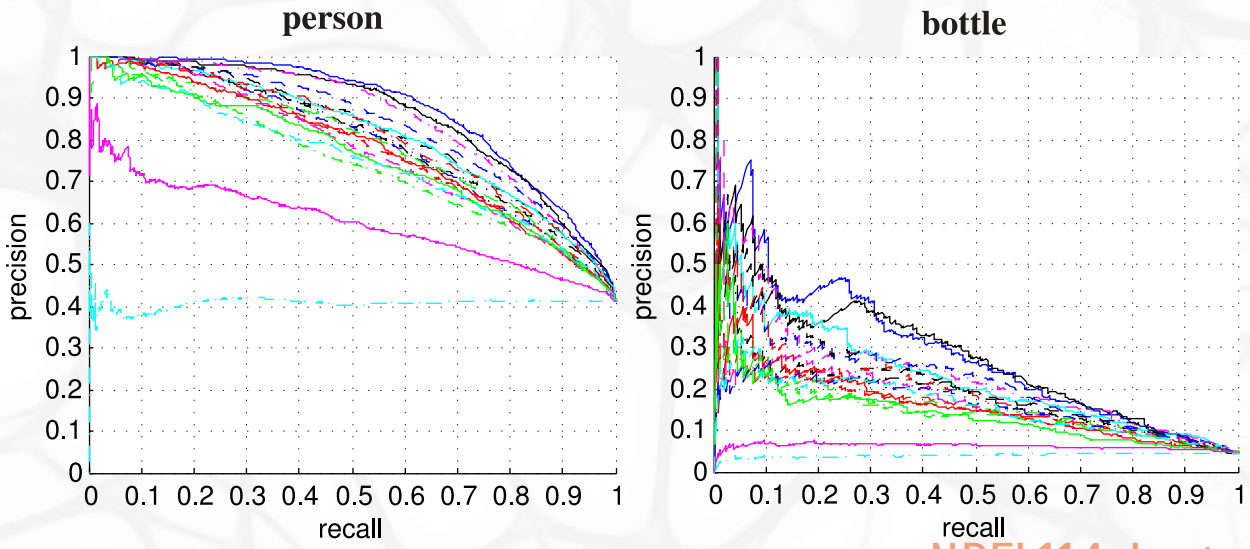
We assume all bounding boxes (or masks) produced by a system have confidence values which can be used to rank them. Then, for a single class, we take the boxes (or masks) in the order of the ranks and generate precision/recall curve, considering a bounding box correct if it has IoU at least 0.5 with any ground-truth box. We define *AP* as an average of precisions for recall levels $0, 0.1, 0.2, \ldots, 1$.

Object Detection Evaluation

Average Precision

Evaluation is performed using *Average Precision* (*AP*).

We assume all bounding boxes (or masks) produced by a system have confidence values which can be used to rank them. Then, for a single class, we take the boxes (or masks) in the order of the ranks and generate precision/recall curve, considering a bounding box correct if it has IoU at least 0.5 with any ground-truth box. We define *AP* as an average of precisions for recall levels $0, 0.1, 0.2, \ldots, 1$.



Faster R-CNN

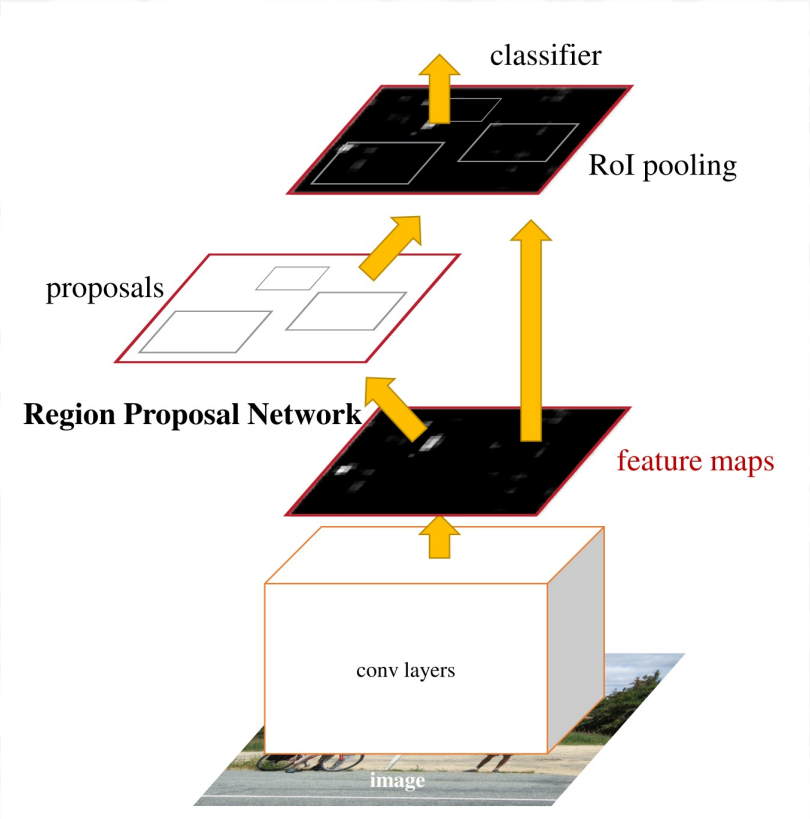
For Fast R-CNN, the most time consuming part is generating the RoIs.

Therefore, Faster R-CNN jointly generates *regions of interest* using a *region proposal network* and performs object detection.

Faster R-CNN

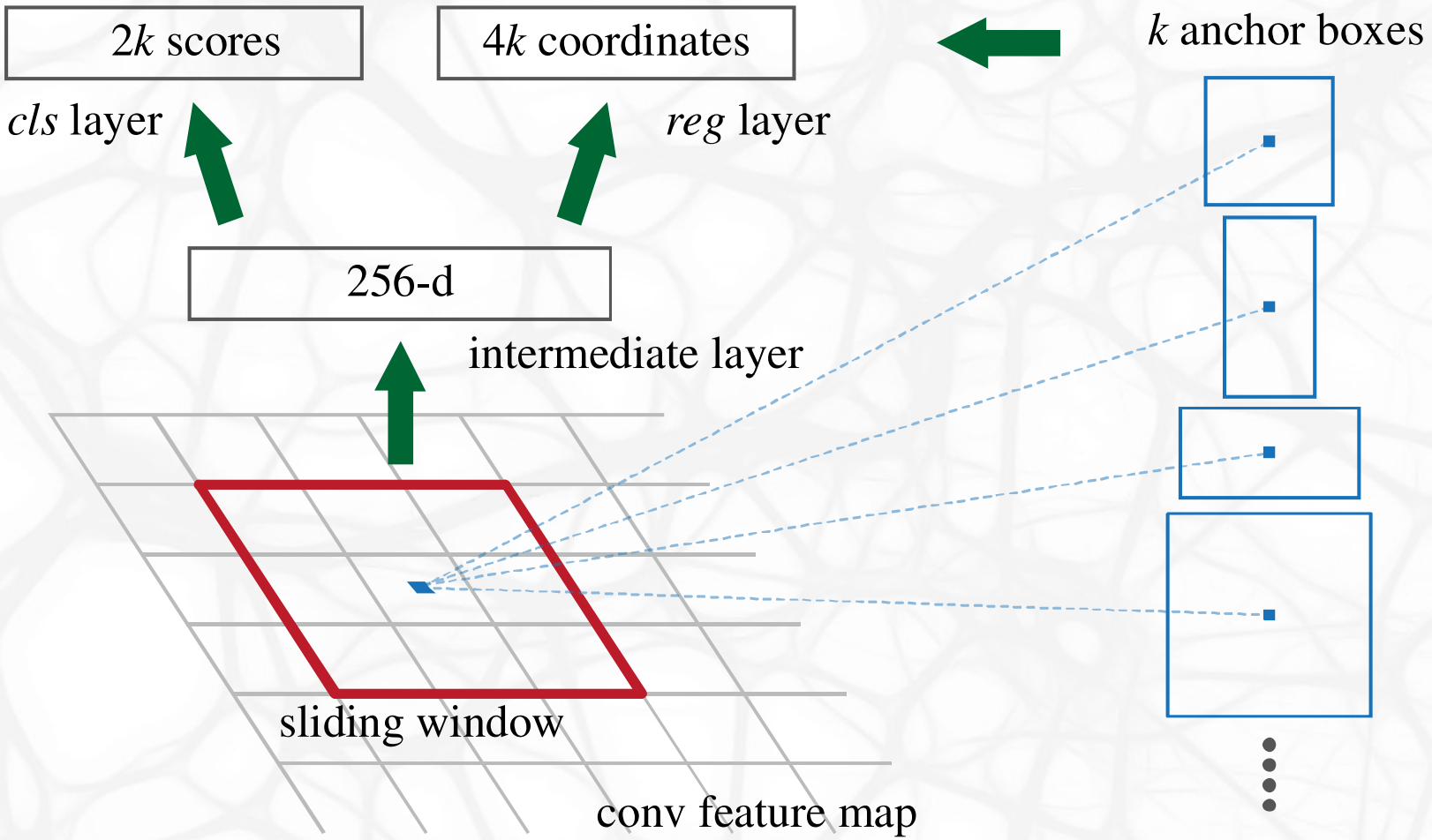
For Fast R-CNN, the most time consuming part is generating the RoIs.

Therefore, Faster R-CNN jointly generates *regions of interest* using a *region proposal network* and performs object detection.



Faster R-CNN

The region proposals are generated using a 3×3 sliding window, with 3 different scales (128^2, 256^2 and 512^2) and 3 aspect ratios ($1:1, 1:2, 2:1$).



Faster R-CNN

Table 3: Detection results on **PASCAL VOC 2007 test set**. The detector is Fast R-CNN and VGG-16. Training data: "07": VOC 2007 trainval, "07+12": union set of VOC 2007 trainval and VOC 2012 trainval. For RPN, the train-time proposals for Fast R-CNN are 2000. †: this number was reported in [2]; using the repository provided by this paper, this result is higher (68.1).

method	# proposals	data	mAP (%)
SS	2000	07	66.9†
SS	2000	07+12	70.0
RPN+VGG, unshared	300	07	68.5
RPN+VGG, shared	300	07	69.9
RPN+VGG, shared	300	07+12	**73.2**
RPN+VGG, shared	300	COCO+07+12	**78.8**

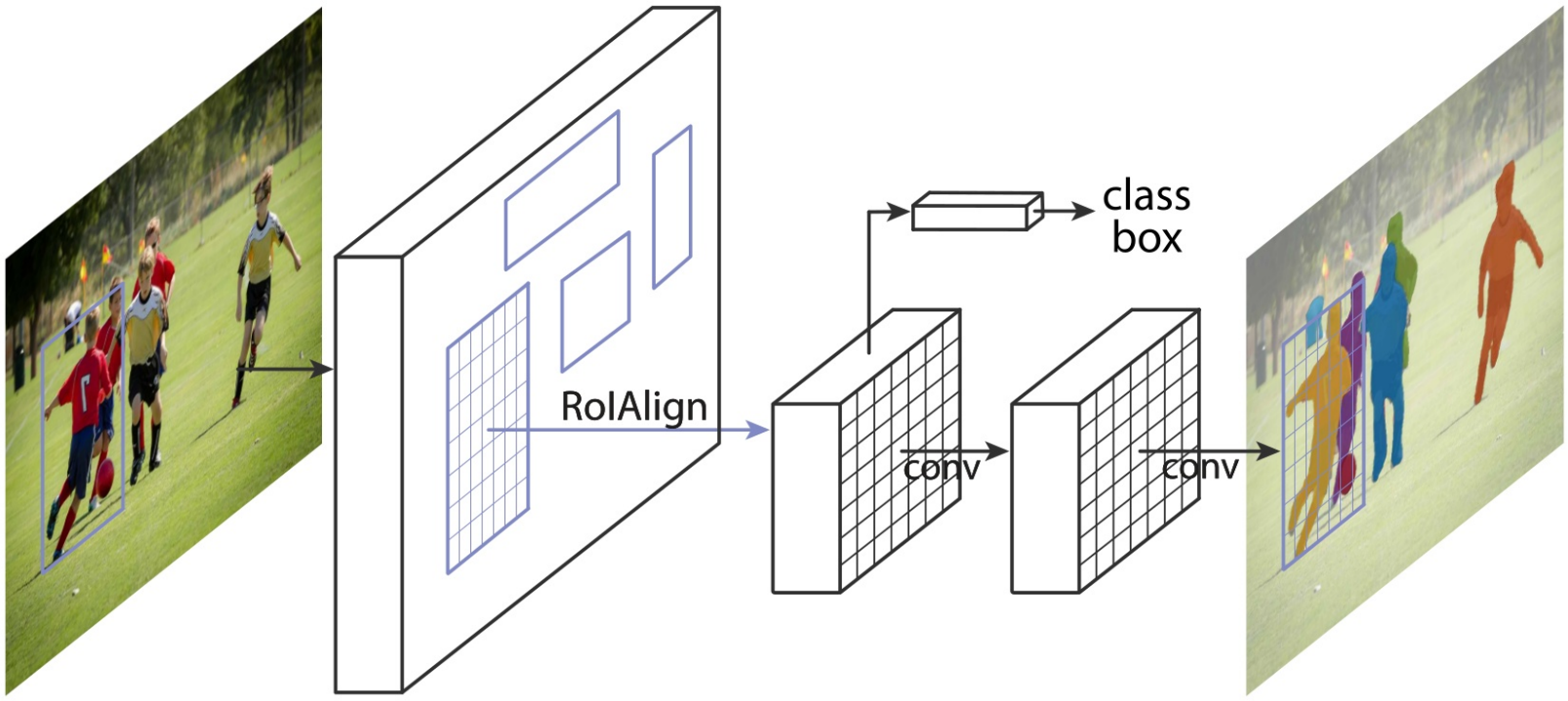
Table 4: Detection results on **PASCAL VOC 2012 test set**. The detector is Fast R-CNN and VGG-16. Training data: "07": VOC 2007 trainval, "07++12": union set of VOC 2007 trainval+test and VOC 2012 trainval. For RPN, the train-time proposals for Fast R-CNN are 2000. †: http://host.robots.ox.ac.uk:8080/anonymous/HZJTQA.html. ‡: http://host.robots.ox.ac.uk:8080/anonymous/YNPLXB.html. §: http://host.robots.ox.ac.uk:8080/anonymous/XEDH10.html.

method	# proposals	data	mAP (%)
SS	2000	12	65.7
SS	2000	07++12	68.4
RPN+VGG, shared†	300	12	67.0
RPN+VGG, shared‡	300	07++12	**70.4**
RPN+VGG, shared§	300	COCO+07++12	**75.9**

Mask R-CNN

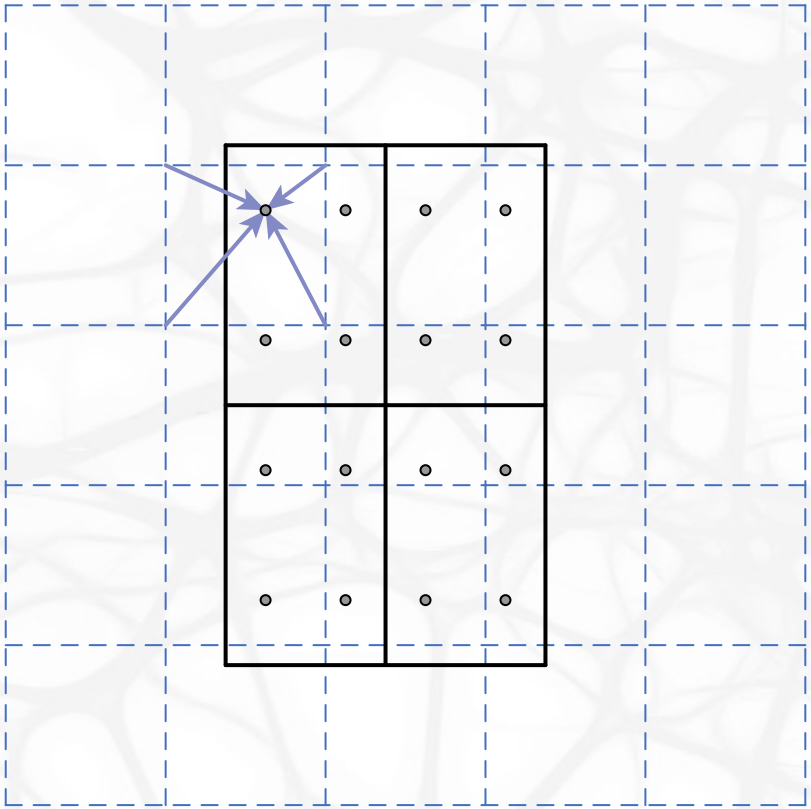
"Straightforward" extension of Faster R-CNN able to produce image segmentation (i.e., masks for every object).

Mask R-CNN



RoIAlign

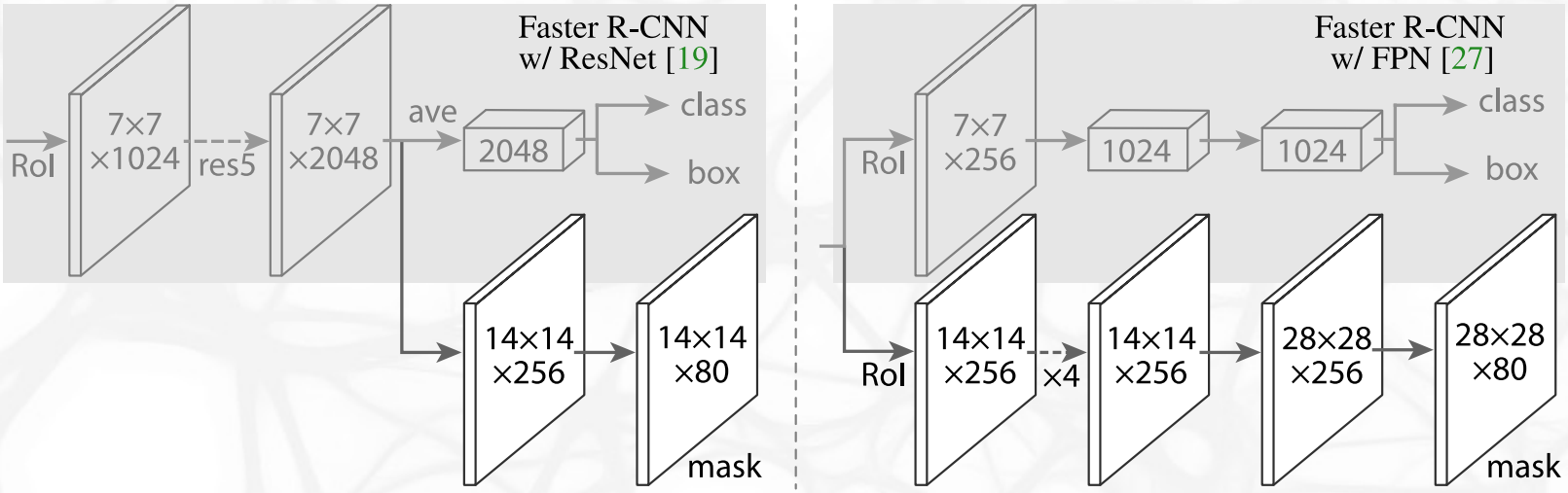
More precise alignment if required for the RoI in order to predict the masks. Therefore, instead of max-pooling used in the RoI pooling, RoIAlign with bilinear interpolation is used instead.



Mask R-CNN

Masks are predicted in a third branch of the object detector.

- Usually higher resolution is needed (14×14 instead of 7×7).
- The masks are predicted for each class separately.
- The masks are predicted using convolutions instead of fully connected layers.



Mask R-CNN

net-depth-features	AP	AP_{50}	AP_{75}
ResNet-50-C4	30.3	51.2	31.5
ResNet-101-C4	32.7	54.2	34.3
ResNet-50-FPN	33.6	55.2	35.3
ResNet-101-FPN	35.4	57.3	37.5
ResNeXt-101-FPN	**36.7**	**59.5**	**38.9**

(a) **Backbone Architecture**: Better backbones bring expected gains: deeper networks do better, FPN outperforms C4 features, and ResNeXt improves on ResNet.

	AP	AP_{50}	AP_{75}
softmax	24.8	44.1	25.1
sigmoid	**30.3**	**51.2**	**31.5**
	+5.5	*+7.1*	*+6.4*

(b) **Multinomial *vs.* Independent Masks** (ResNet-50-C4): *Decoupling* via per-class binary masks (sigmoid) gives large gains over multinomial masks (softmax).

	align?	bilinear?	agg.	AP	AP_{50}	AP_{75}
RoIPool [12]			max	26.9	48.8	26.4
RoIWarp [10]		✓	max	27.2	49.2	27.1
		✓	ave	27.1	48.9	27.1
RoIAlign	✓	✓	max	**30.2**	**51.0**	**31.8**
	✓	✓	ave	**30.3**	**51.2**	**31.5**

(c) **RoIAlign** (ResNet-50-C4): Mask results with various RoI layers. Our RoIAlign layer improves AP by ~3 points and AP_{75} by ~5 points. Using proper alignment is the only factor that contributes to the large gap between RoI layers.

	AP	AP_{50}	AP_{75}	AP^{bb}	AP^{bb}_{50}	AP^{bb}_{75}
RoIPool	23.6	46.5	21.6	28.2	52.7	26.9
RoIAlign	**30.9**	**51.8**	**32.1**	**34.0**	**55.3**	**36.4**
	+7.3	*+ 5.3*	*+10.5*	*+5.8*	*+2.6*	*+9.5*

(d) **RoIAlign** (ResNet-50-**C5**, *stride 32*): Mask-level and box-level AP using *large-stride* features. Misalignments are more severe than with stride-16 features (Table 2c), resulting in big accuracy gaps.

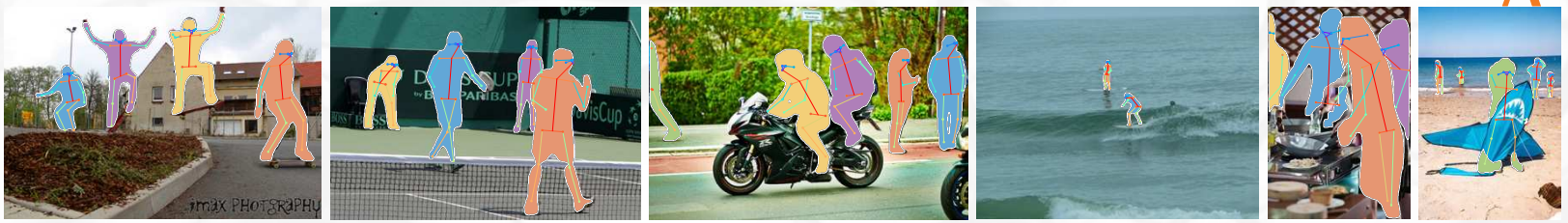
	mask branch	AP	AP_{50}	AP_{75}
MLP	fc: 1024→1024→80·28^2	31.5	53.7	32.8
MLP	fc: 1024→1024→1024→80·28^2	31.5	54.0	32.6
FCN	conv: 256→256→256→256→256→80	**33.6**	**55.2**	**35.3**

(e) **Mask Branch** (ResNet-50-FPN): Fully convolutional networks (FCN) *vs.* multi-layer perceptrons (MLP, fully-connected) for mask prediction. FCNs improve results as they take advantage of explicitly encoding spatial layout.

Table 2. **Ablations**. We train on `trainval35k`, test on `minival`, and report *mask* AP unless otherwise noted.

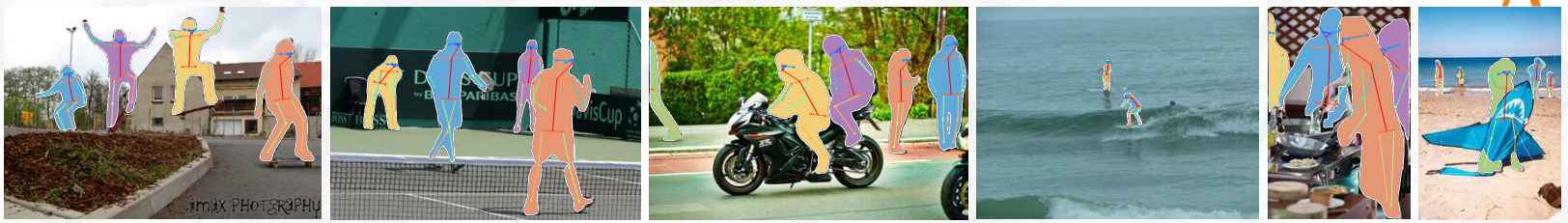
Mask R-CNN – Human Pose Estimation

Mask R-CNN – Human Pose Estimation



- Testing applicability of Mask R-CNN architecture.
- Keypoints (e.g., left shoulder, right elbow, …) are detected as independent one-hot masks of size 56×56 with $\mathrm{softmax}$ output function.

Mask R-CNN – Human Pose Estimation



- Testing applicability of Mask R-CNN architecture.
- Keypoints (e.g., left shoulder, right elbow, …) are detected as independent one-hot masks of size 56×56 with $\mathrm{softmax}$ output function.

	AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}	AP^{kp}_{M}	AP^{kp}_{L}
CMU-Pose+++ [6]	61.8	84.9	67.5	57.1	68.2
G-RMI [32]†	62.4	84.0	68.5	**59.1**	68.1
Mask R-CNN, keypoint-only	62.7	87.0	68.4	57.4	71.1
Mask R-CNN, keypoint & mask	**63.1**	**87.3**	**68.7**	57.8	**71.4**

Batch Normalization

Neuron value is normalized across the minibatch, and in case of CNN also across all positions.

Normalization

Batch Normalization

Neuron value is normalized across the minibatch, and in case of CNN also across all positions.

Layer Normalization

Neuron value is normalized across the layer.

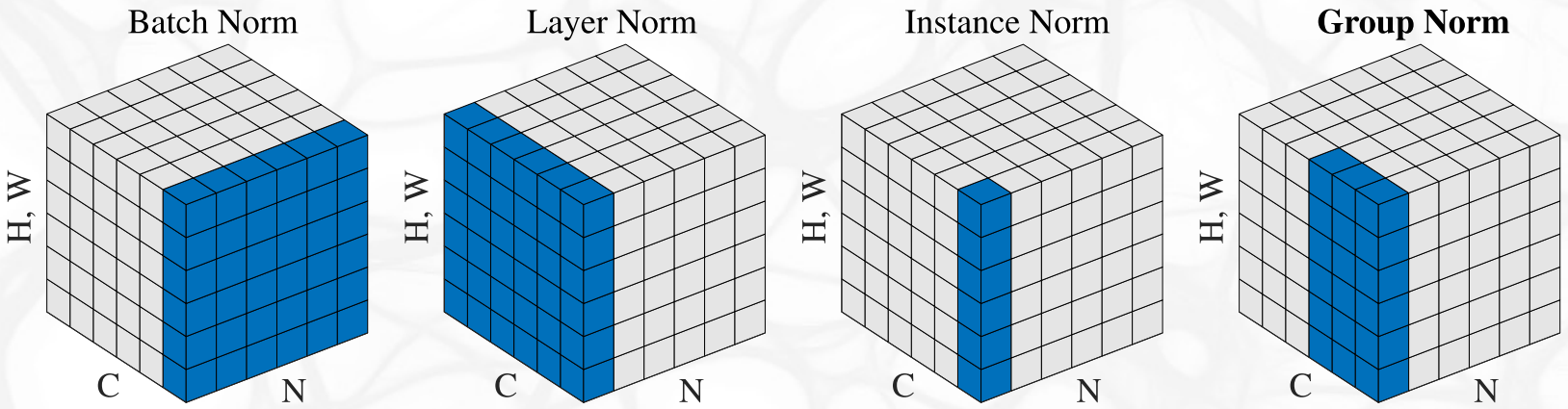
Normalization

Batch Normalization

Neuron value is normalized across the minibatch, and in case of CNN also across all positions.

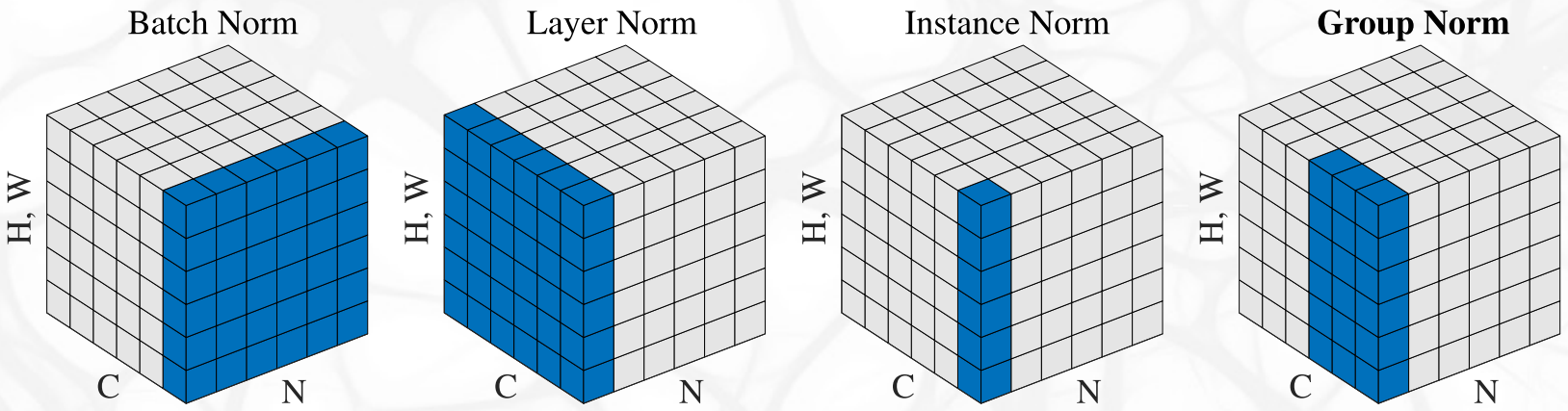
Layer Normalization

Neuron value is normalized across the layer.



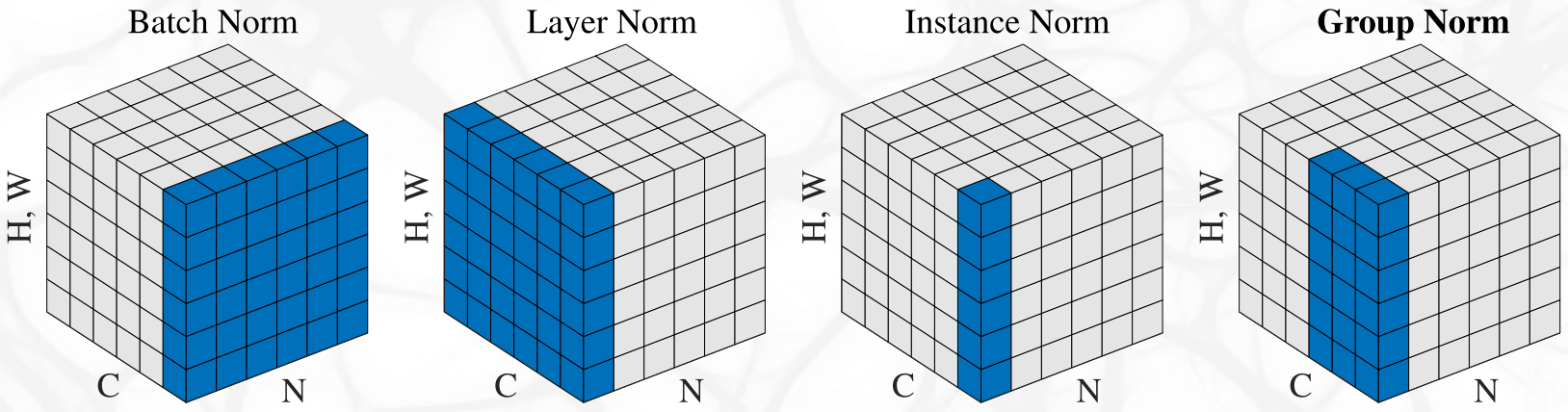
Group Normalization

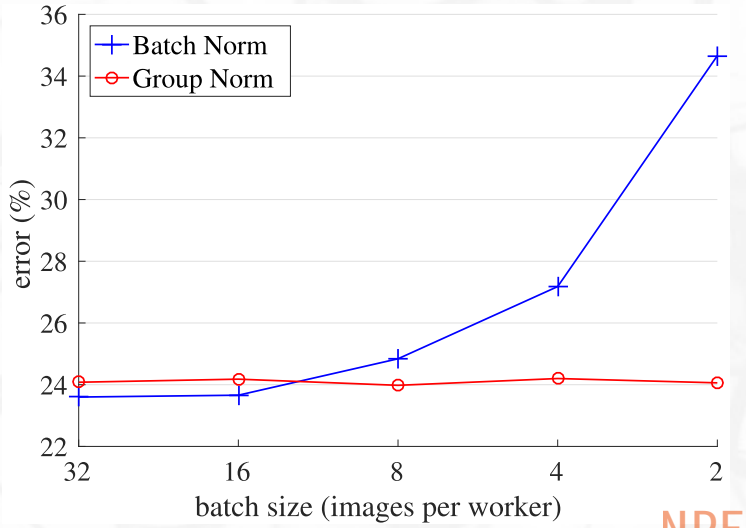
Group Normalization is analogous to Layer normalization, but the channels are normalized in groups (by default, $G = 32$).



Group Normalization

Group Normalization is analogous to Layer normalization, but the channels are normalized in groups (by default, $G = 32$).





Group Normalization

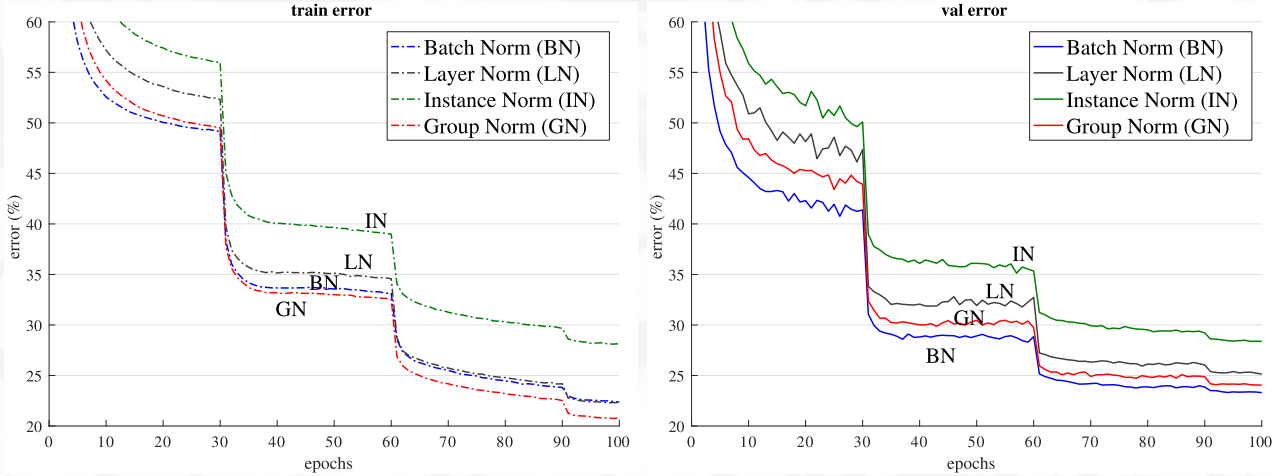


Figure 4. Comparison of error curves with a batch size of 32 images/GPU. We show the ImageNet training error (left) and validation error (right) vs. numbers of training epochs. The model is ResNet-50.

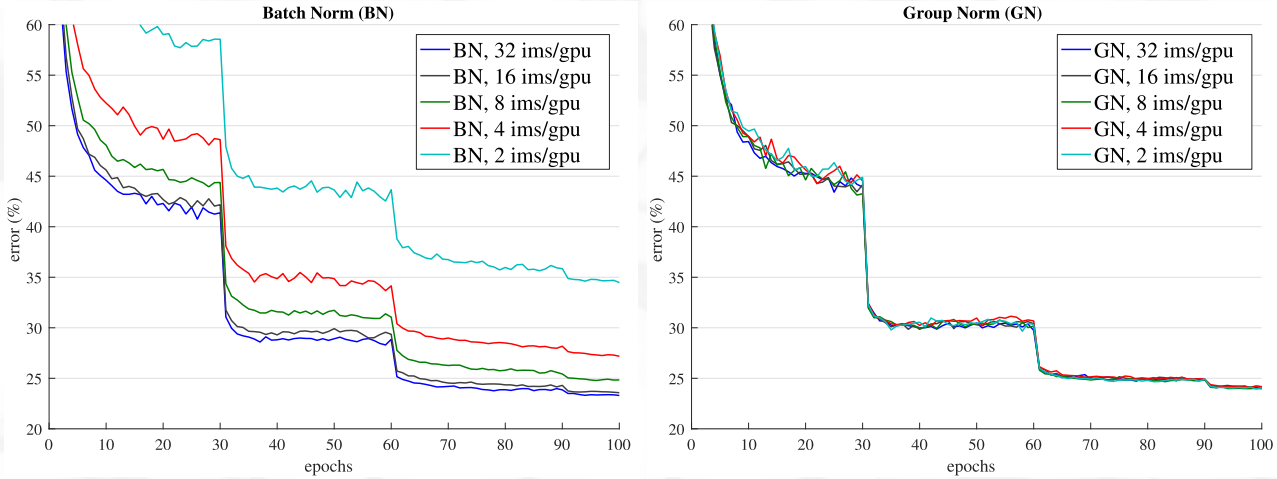


Figure 5. Sensitivity to batch sizes: ResNet-50's validation error of BN (left) and GN (right) trained with 32, 16, 8, 4, and 2 images/GPU.

Group Normalization

backbone	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
BN^*	37.7	57.9	40.9	32.8	54.3	34.7
GN	**38.8**	**59.2**	**42.2**	**33.6**	**55.9**	**35.4**

Table 4. **Detection and segmentation results in COCO**, using Mask R-CNN with **ResNet-50 C4**. BN^* means BN is frozen.

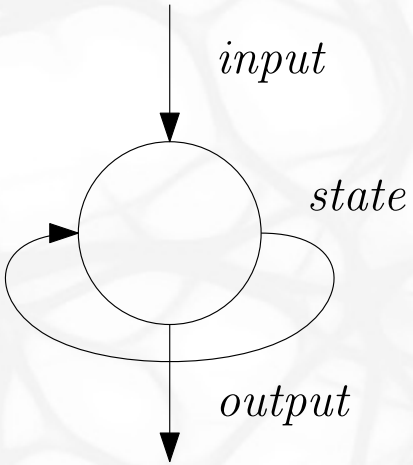
backbone	box head	AP^{bbox}	AP^{bbox}_{50}	AP^{bbox}_{75}	AP^{mask}	AP^{mask}_{50}	AP^{mask}_{75}
BN^*	-	38.6	59.5	41.9	34.2	56.2	36.1
BN^*	GN	39.5	60.0	43.2	34.4	56.4	**36.3**
GN	GN	**40.0**	**61.0**	**43.3**	**34.8**	**57.3**	**36.3**

Table 5. **Detection and segmentation results in COCO**, using Mask R-CNN with **ResNet-50 FPN** and a 4conv1fc bounding box head. BN^* means BN is frozen.

Recurrent Neural Networks

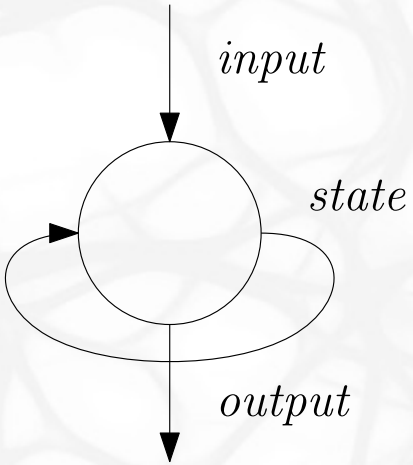
Recurrent Neural Networks

Single RNN cell

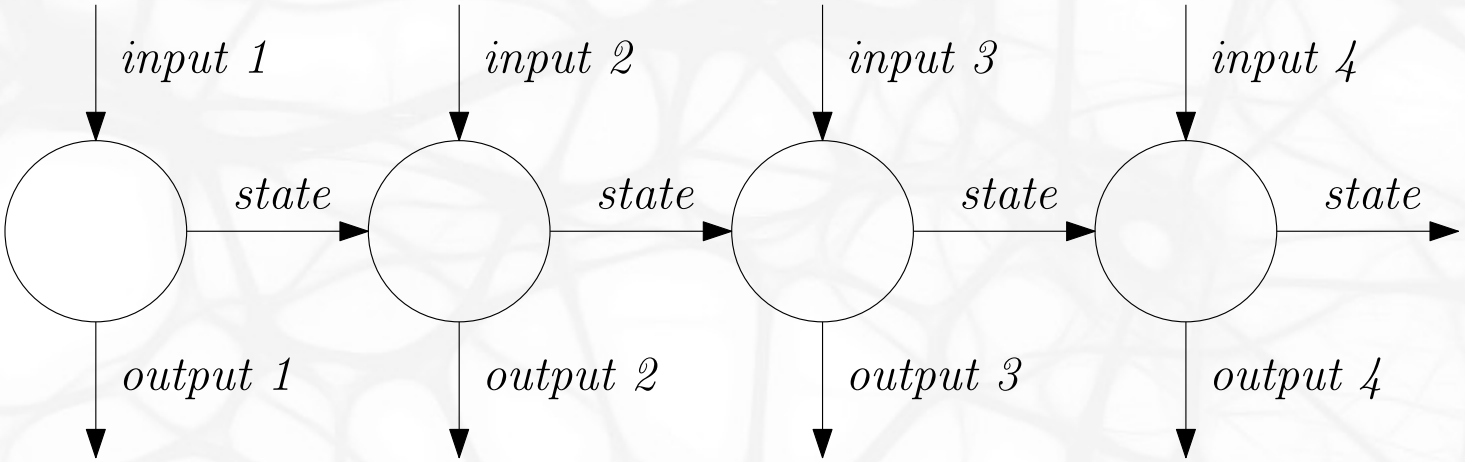


Recurrent Neural Networks

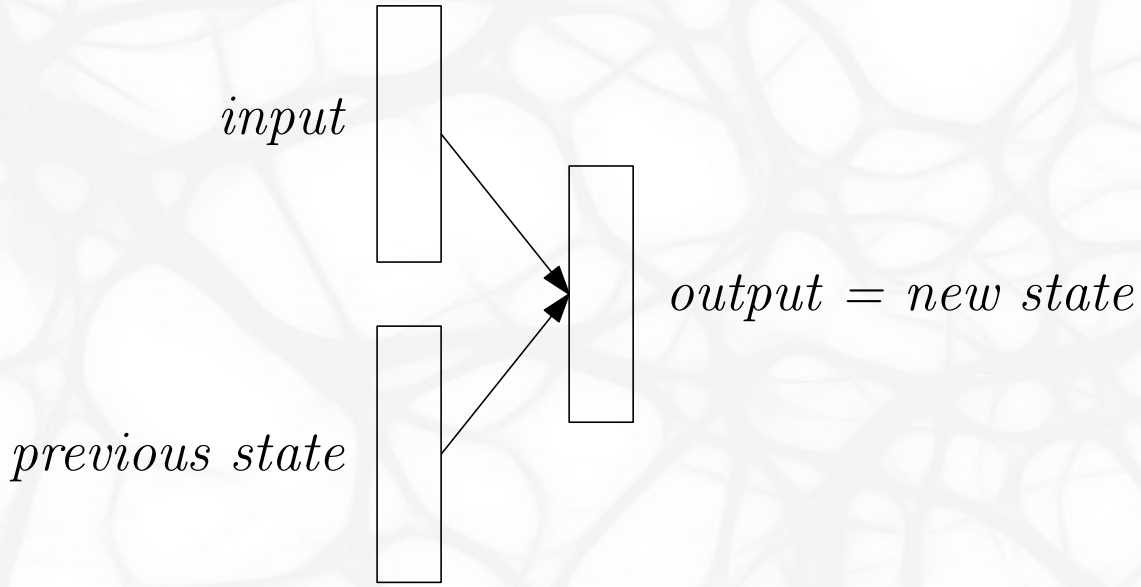
Single RNN cell



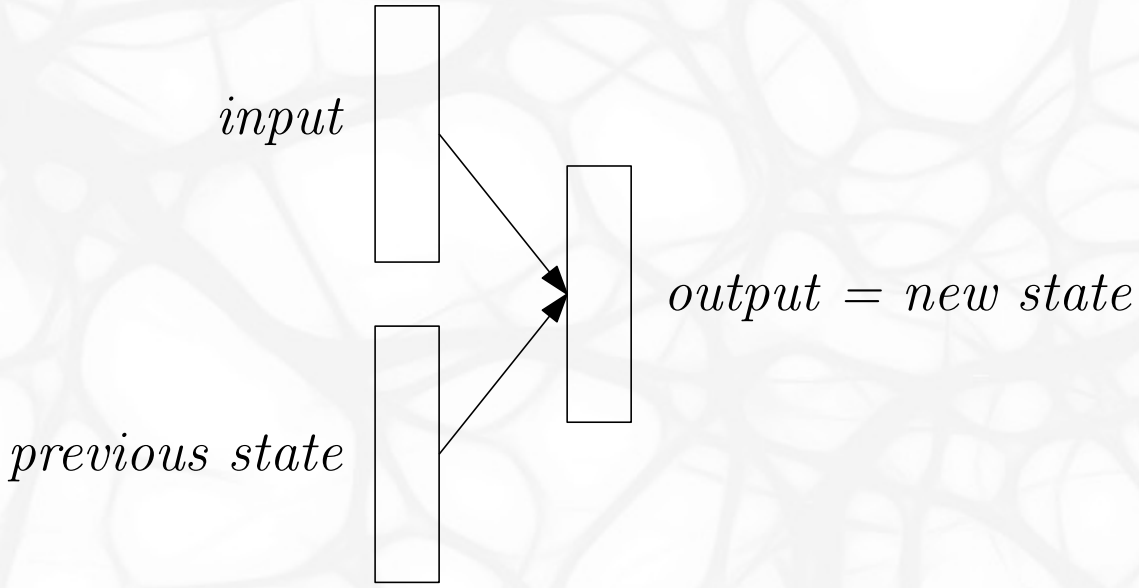
Unrolled RNN cells



Basic RNN Cell



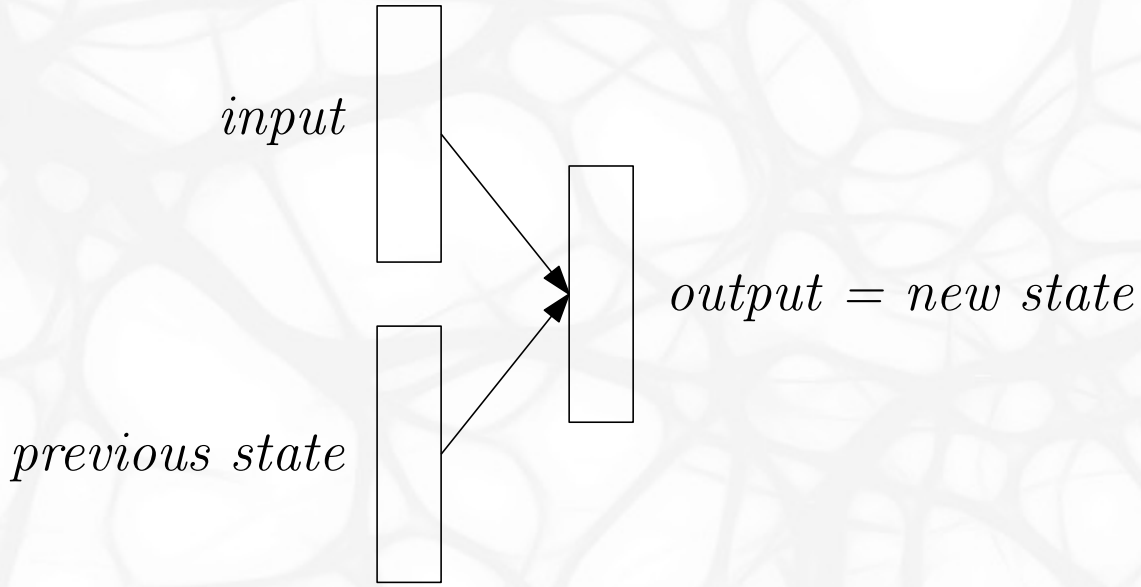
Basic RNN Cell



Given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{s}^{(t-1)}$, the new state is computed as

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}).$$

Basic RNN Cell



Given an input $\boldsymbol{x}^{(t)}$ and previous state $\boldsymbol{s}^{(t-1)}$, the new state is computed as

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}).$$

One of the simplest possibilities is

$$\boldsymbol{s}^{(t)} = \tanh(\boldsymbol{U}\boldsymbol{s}^{(t-1)} + \boldsymbol{V}\boldsymbol{x}^{(t)} + \boldsymbol{b}).$$

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*).

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*).

If we simplify the recurrence of states to

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}\boldsymbol{s}^{(t-1)},$$

we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}^t\boldsymbol{s}^{(0)}.$$

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*).

If we simplify the recurrence of states to

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}\boldsymbol{s}^{(t-1)},$$

we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}^t\boldsymbol{s}^{(0)}.$$

If U has eigenvalue decomposition of $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{s}^{(0)}.$$

Basic RNN Cell

Basic RNN cells suffer a lot from vanishing/exploding gradients (*the challenge of long-term dependencies*).

If we simplify the recurrence of states to

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}\boldsymbol{s}^{(t-1)},$$

we get

$$\boldsymbol{s}^{(t)} = \boldsymbol{U}^t\boldsymbol{s}^{(0)}.$$

If U has eigenvalue decomposition of $\boldsymbol{U} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{-1}$, we get

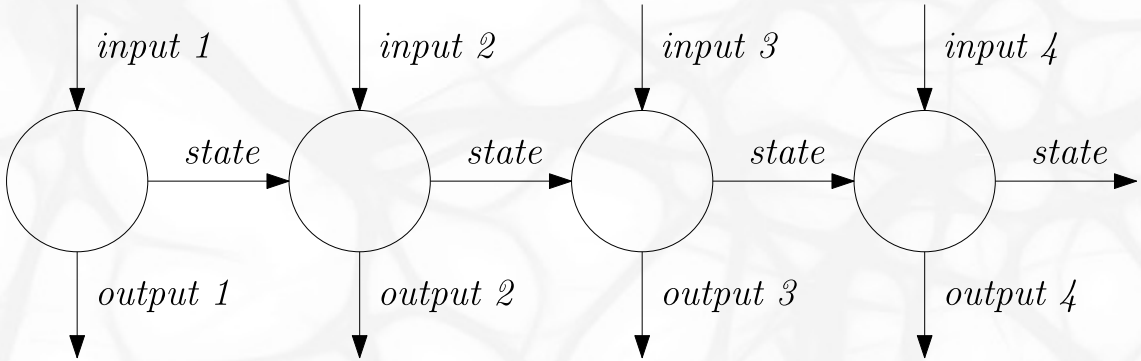
$$\boldsymbol{s}^{(t)} = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^{-1}\boldsymbol{s}^{(0)}.$$

Several more complex RNN cell variants have been proposed, which alleviate this issue to some degree, namely **LSTM** and **GRU**. They will be introduced in the next lecture.

Basic RNN Applications

Sequence Element Classification

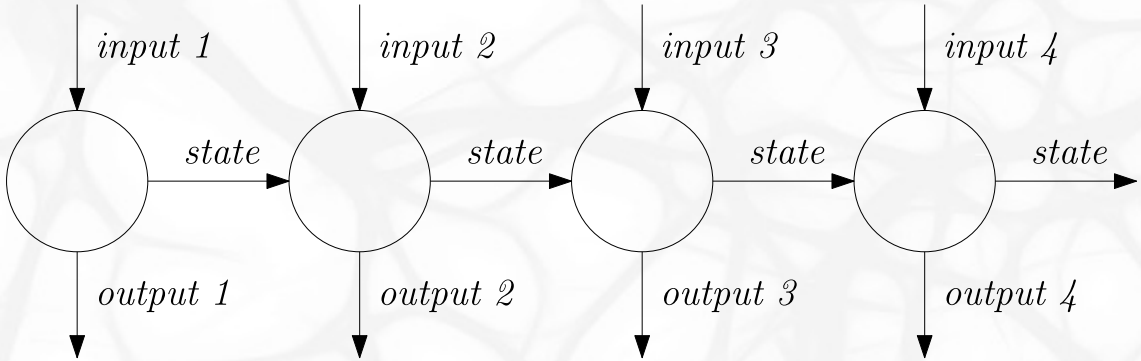
Use outputs for individual elements.



Basic RNN Applications

Sequence Element Classification

Use outputs for individual elements.



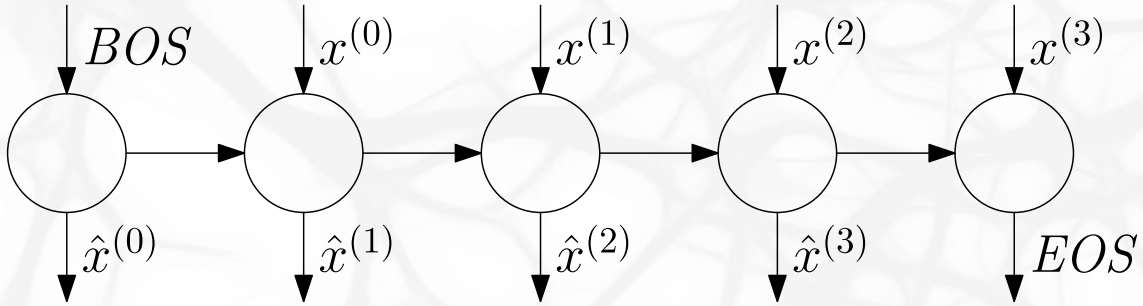
Sequence Representation

Use state after processing the whole sequence (alternatively, take output of the last element).

Basic RNN Applications

Sequence Prediction

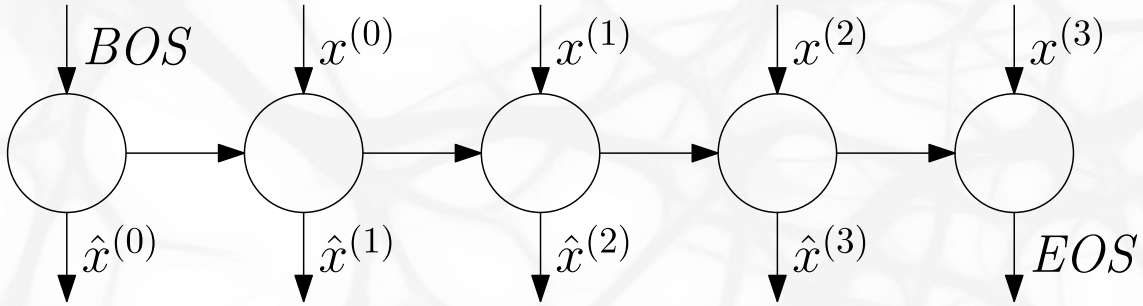
During training, predict next sequence element.



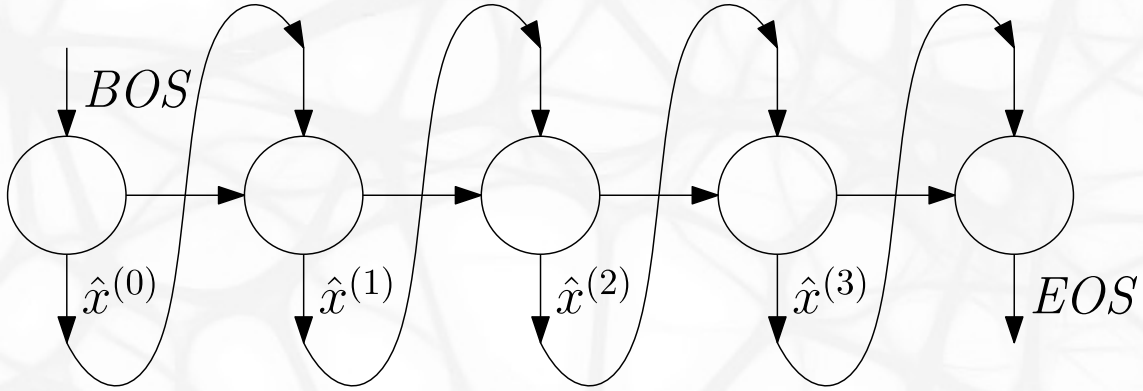
Basic RNN Applications

Sequence Prediction

During training, predict next sequence element.



During inference, use predicted elements as further inputs.



TF – Functional vs Object Layers

So far, we have been using functional layers interface.

```
tf.layers.dense(inputs, units, activation=None, ...)
```

Generally, each functional call creates a new layer (although it is possible to employ `name` and `reuse` parameters and share layers of the same name).

TF – Functional vs Object Layers

So far, we have been using functional layers interface.

```
tf.layers.dense(inputs, units, activation=None, ...)
```

Generally, each functional call creates a new layer (although it is possible to employ `name` and `reuse` parameters and share layers of the same name).

TensorFlow also supports (newer) object interface.

```
layer = tf.layers.Dense(units, activation=None, ...)

y1, y2 = layer(x1), layer(x2)
```

TF – RNN Cells

RNN cells are provided only through the object interface.

- `tf.nn.rnn_cell.RNNCell`: abstract superclass
- `tf.nn.rnn_cell.BasicRNNCell`: the basic RNN cell described earlier
- `tf.nn.rnn_cell.{BasicLSTMCell,LSTMCell}`: the LSTM cells
- `tf.nn.rnn_cell.GRUCell`: the GRU cell
- various wrappers, and also additional types in `tf.contrib.rnn`

TF – RNN Cells

RNN cells are provided only through the object interface.

- `tf.nn.rnn_cell.RNNCell`: abstract superclass
- `tf.nn.rnn_cell.BasicRNNCell`: the basic RNN cell described earlier
- `tf.nn.rnn_cell.{BasicLSTMCell,LSTMCell}`: the LSTM cells
- `tf.nn.rnn_cell.GRUCell`: the GRU cell
- various wrappers, and also additional types in `tf.contrib.rnn`

```
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(units,
                                       activation=None, ...)

output, state = rnn_cell(input, state)

rnn_cell.zero_state(batch_size, dtype)
```

TF – Processing Sequences with RNN

tf.nn.static_rnn

```
outputs, state = tf.nn.static_rnn(cell,
                                  inputs,
                                  initial_state=None,
                                  ...)
```

TF – Processing Sequences with RNN

tf.nn.static_rnn

```
outputs, state = tf.nn.static_rnn(cell,
                                  inputs,
                                  initial_state=None,
                                  ...)
```

The `inputs` is a list of tensors, each of shape *(batchsize, inputsize)*, and analogously for `outputs`. The `state` is the last state.

The length of `inputs` and `outputs` must be statically known.

TF – Processing Sequences with RNN

tf.nn.dynamic_rnn

```
outputs, state = tf.nn.dynamic_rnn(cell,
                                   inputs,
                                   initial_state=None,
                                   time_major=False,
                                   ...)
```

TF – Processing Sequences with RNN

tf.nn.dynamic_rnn

```
outputs, state = tf.nn.dynamic_rnn(cell,
                                   inputs,
                                   initial_state=None,
                                   time_major=False,
                                   ...)
```

- If `time_major` is `False` (default for consistency with other operations), then `inputs` is a tensor with shape *(batchsize, maxtime, inputsize)*, and analogously for `outputs`.
- If `time_major` is `True` (faster variant), then `inputs` is a tensor with shape *(maxtime, batchsize, inputsize)*, and analogously for `outputs`.

The `state` is again the last state.

The length of the sequences do not need to be known statically (a `while` cycle is added to the computational graph).