Lecture 15: RNNs + Object Detection and Segmentation

Announcements:

- Remaining schedule: Today: RNNs + object detection, 3/11: object detection + adversarial examples, 3/13: adversarial + overview.
- The project and its accompanying data have been uploaded to Bruin Learn. It is due **March 15, 2024** (Friday of Week 10).
    - You will be allowed to use PyTorch, Keras, or other deep learning libraries for the project.
- Midterm regrades are due by Monday, 3/11, at 11:59pm.

Vanilla RNN architecture

state at time t

Markov assumption $s_t = f(s_{t-1}, x_t)$

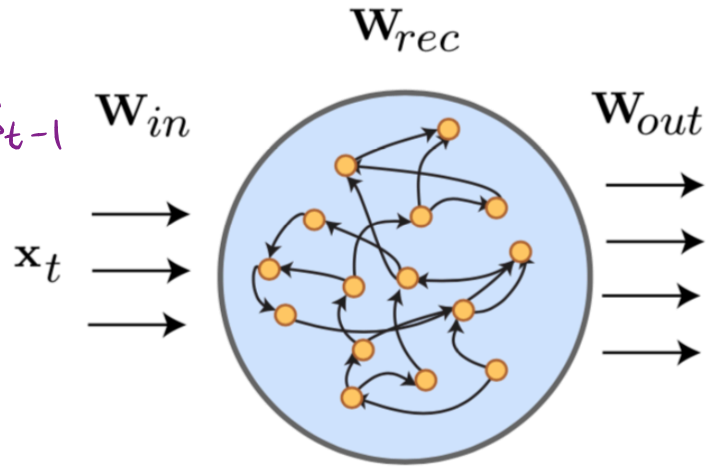
$= g(s_{t-1}, s_{t-2}, x_t)$

$h_t = \mathrm{relu}\left(W_{rec} h_{t-1} + W_{in} x_t + b\right)$

At a high-level, the RNN can be diagrammed as follows:

$h_t \in \mathbb{R}^D \quad \forall t$

$s_t \perp\!\!\!\perp s_{t-2} \mid s_{t-1}$



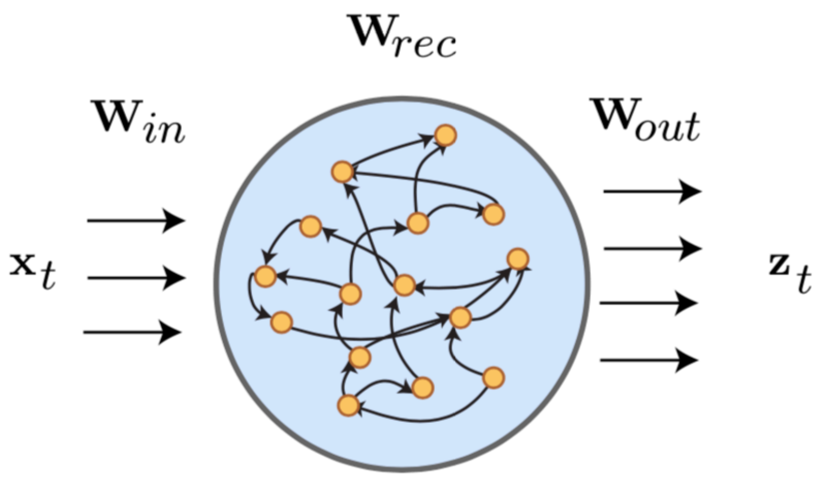
$\in \mathbb{R}^{26}$ $\quad 26 \times D \quad D$

$\mathbf{z}_t = W_{out} h_t + b_{out}$

The RNN has three major components:

- $\mathbf{W}_{\rm in}$: An input at time t, denoted $\mathbf{x}_t$, is transformed via $\mathbf{W}_{\rm in}$ onto artificial neurons, whose activations are $\mathbf{h}_t$.
- $\mathbf{W}_{\rm rec}$: Each artificial neuron is the network is denoted by an orange circle, and these artificial neurons have recurrent connections. recurrent connections are defined by the matrix $\mathbf{W}_{\rm rec}$.
- $\mathbf{W}_{\rm out}$: Finally, the artificial neuron activations are mapped linearly to the output $\mathbf{z}_t$ through the matrix $\mathbf{W}_{\rm out}$.

What do we need to train an RNN?



Let’s take stock of what we know:

- We know the RNN equations, and we can define a loss function.
    - **So we know how to do a forward pass and calculate a loss.**

- In general, we know how to do optimization (i.e., with SGD and your favorite optimizer on top of that, e.g., Adam or RMSprop).

- Do we know how to take gradients of the weight matrices?

- Is there any problem in applying backpropagation as in feedforward networks (e.g., CNNs, FC nets) to RNNs?

Multiplicative RNN, $W_{rec}(t)$

Key insight: unroll the computational graph

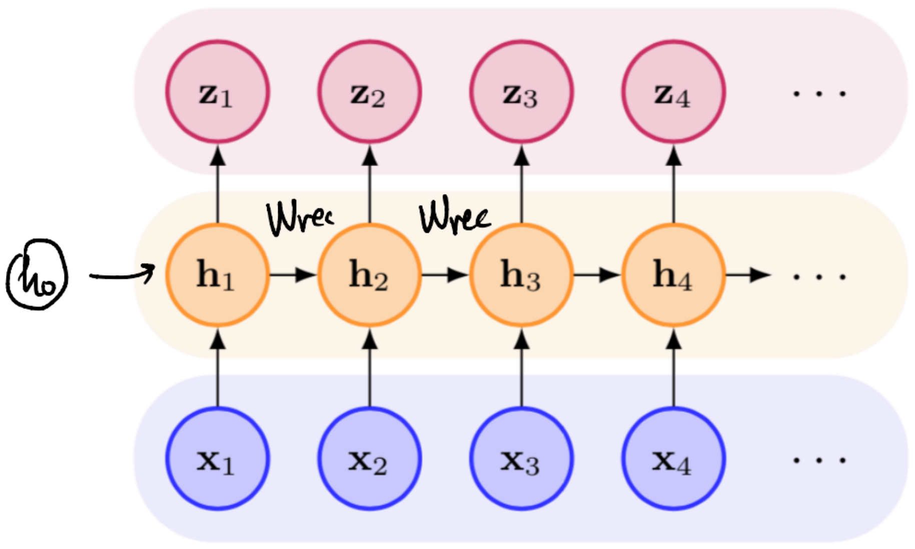
RNN training (cont.)

$h_t = \mathrm{relu}(W_{rec} h_{t-1} + W_{in} x_t)$

To get around this confound, we consider the RNN as a computational graph through time.

$\underbrace{h_0, x_1}_{}$

$\underbrace{h_1, x_2}_{h_2}$



$t=1$ $t=2$ $t=3$

$\longrightarrow$ time

"Backpropagation through time" BPTT

Key insight: unroll the computational graph

$h_t \in \mathbb{R}^4$

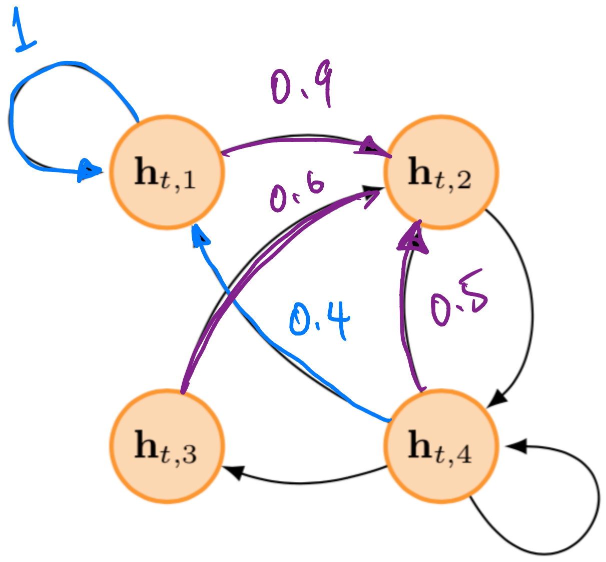
$h_{t+1} = \text{relu}\left(W_{rec} \, h_t \right)$

RNN training (cont.)

For example, consider a matrix $\mathbf{W}_{\text{rec}}$ that looks like the following:

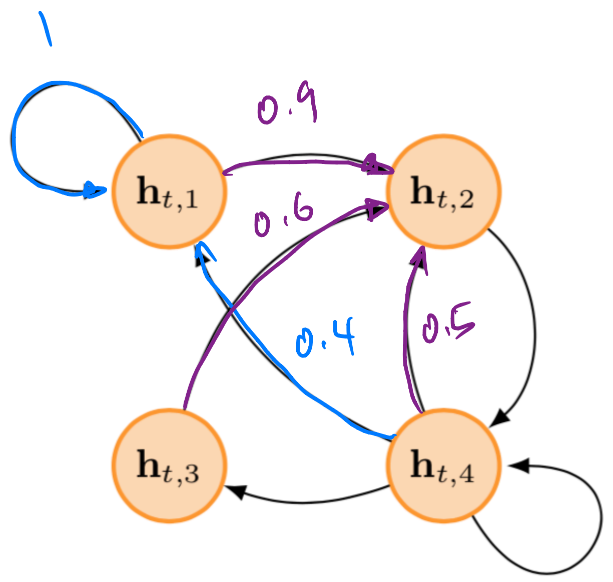
$$h_{t+1} = \begin{bmatrix} h_{t+1,1} \\ h_{t+1,2} \\ h_{t+1,3} \\ h_{t+1,4} \end{bmatrix} = \overset{\text{relu}}{\cancel{\mathbf{W}_{\text{rec}} =}} \left(\begin{bmatrix} 1 & 0 & 0 & 0.4 \\ 0.9 & 0 & 0.6 & 0.5 \\ 0 & 0 & 0 & 0.3 \\ 0 & 0.8 & 0 & 0.7 \end{bmatrix} \begin{bmatrix} h_{t,1} \\ h_{t,2} \\ h_{t,3} \\ h_{t,4} \end{bmatrix} \right)$$

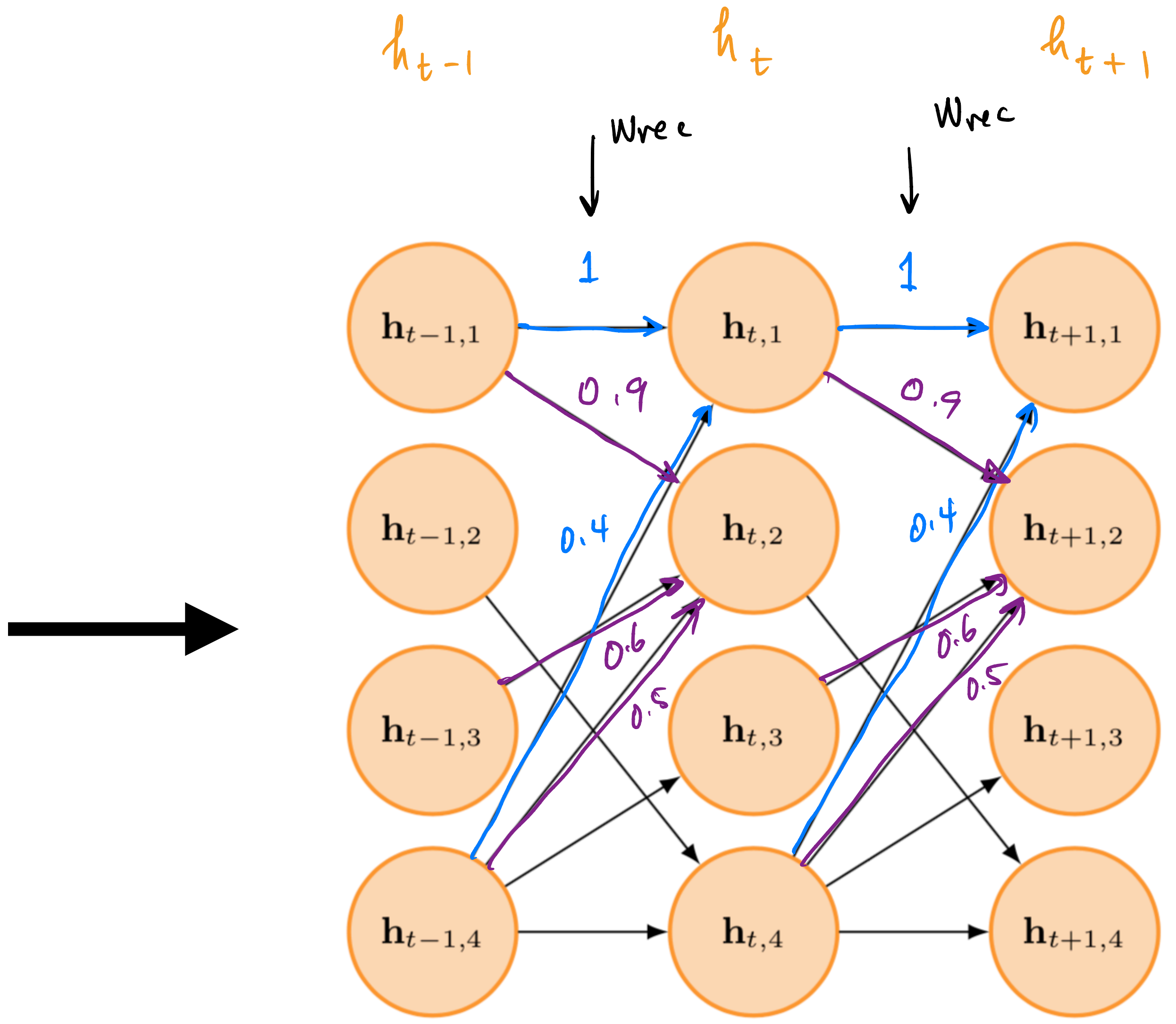
This RNN looks like the following.



Multiplicative RNNs $W_{rec}(t)$

Key insight: unroll the computational graph



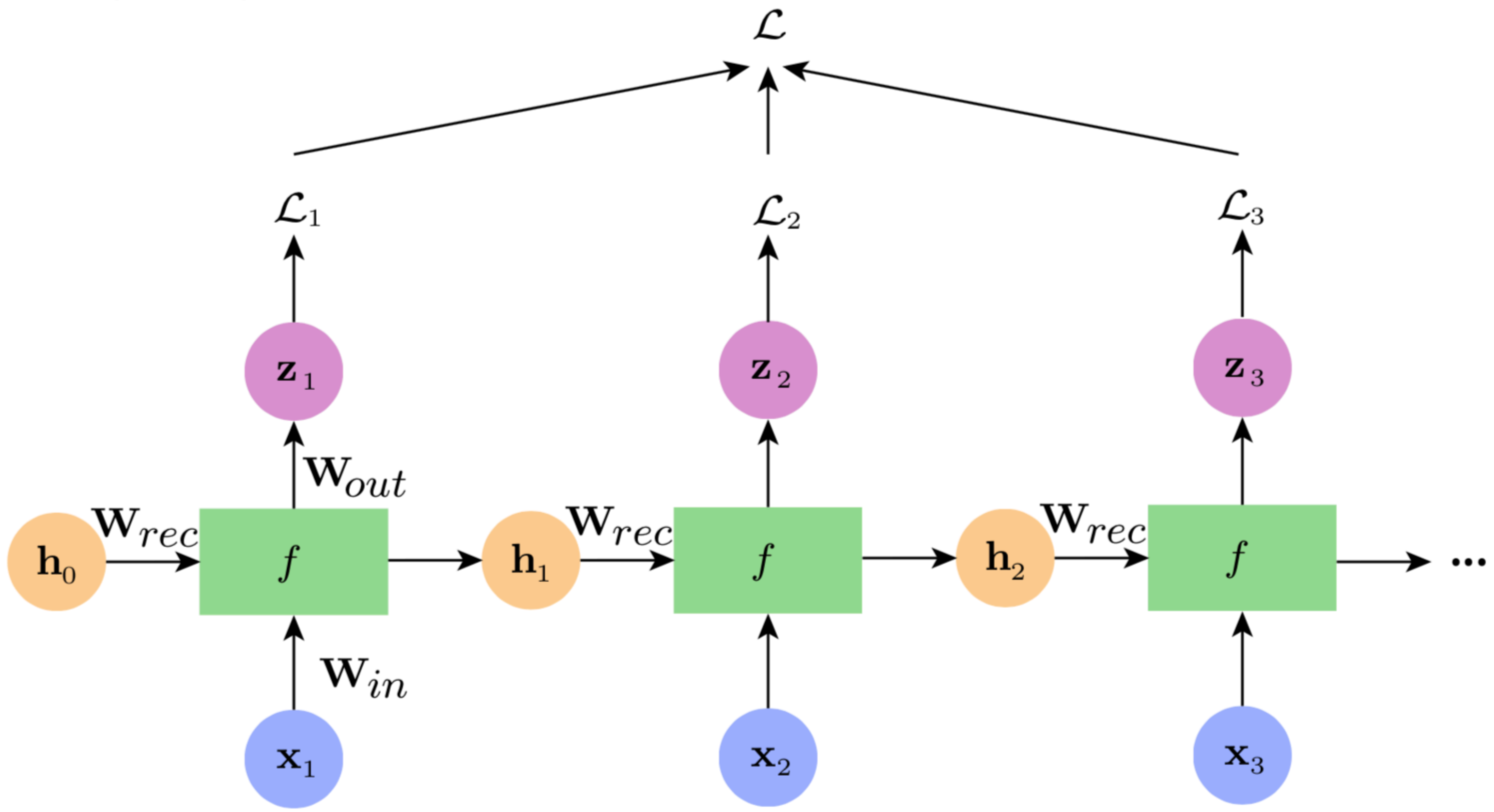


$$W_{rec}^T \left(W_{rec}^T\right) \frac{\partial \mathcal{L}}{\partial h_{t+1}} \quad \longleftarrow \quad W_{rec}^T \frac{\partial \mathcal{L}}{\partial h_{t+1}} \quad \longleftarrow \quad \frac{\partial \mathcal{L}}{\partial h_{t+1}}$$

Key insight: unroll the computational graph

RNN training (cont.)

To train the network, we calculate gradients on the unrolled graph. Doing backpropagation through the unrolled graph is called *backpropagation through time* (BPTT).



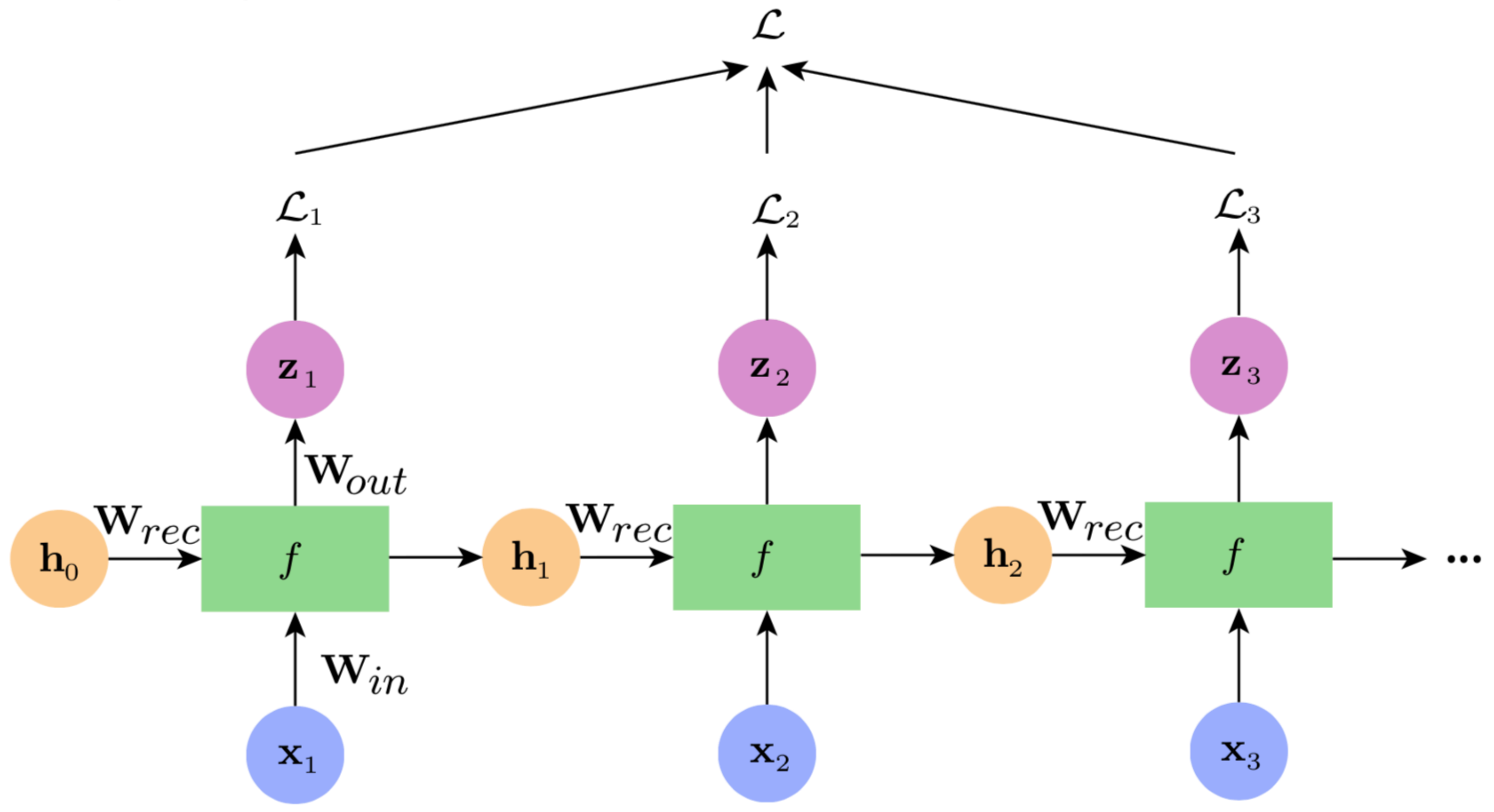
Note that sometimes, one may only care about the last loss $\mathcal{L}_T$, in which case $L_t = 0$ for $t < T$.

$t=1$ $t=2$ $t=3$

Key insight: unroll the computational graph

RNN training (cont.)

To train the network, we calculate gradients on the unrolled graph. Doing backpropagation through the unrolled graph is called *backpropagation through time* (BPTT).



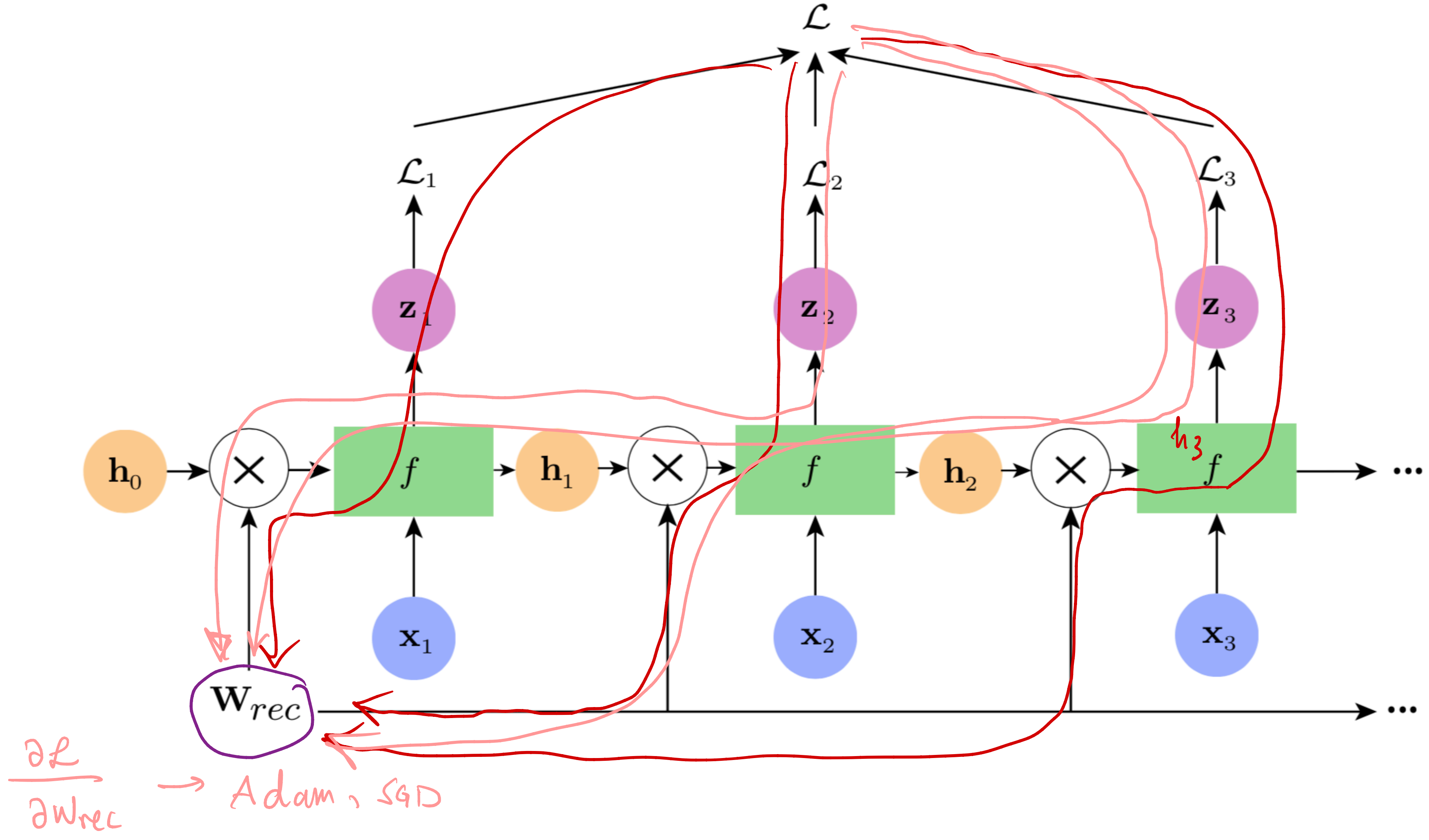
Note that sometimes, one may only care about the last loss $\mathcal{L}_T$, in which case $L_t = 0$ for $t < T$.

Key insight: unroll the computational graph

RNN training (cont.)

This graph can be redrawn as follows:

$\mathcal{L} = \mathcal{L}_1 + \mathcal{L}_2 + \mathcal{L}_3$

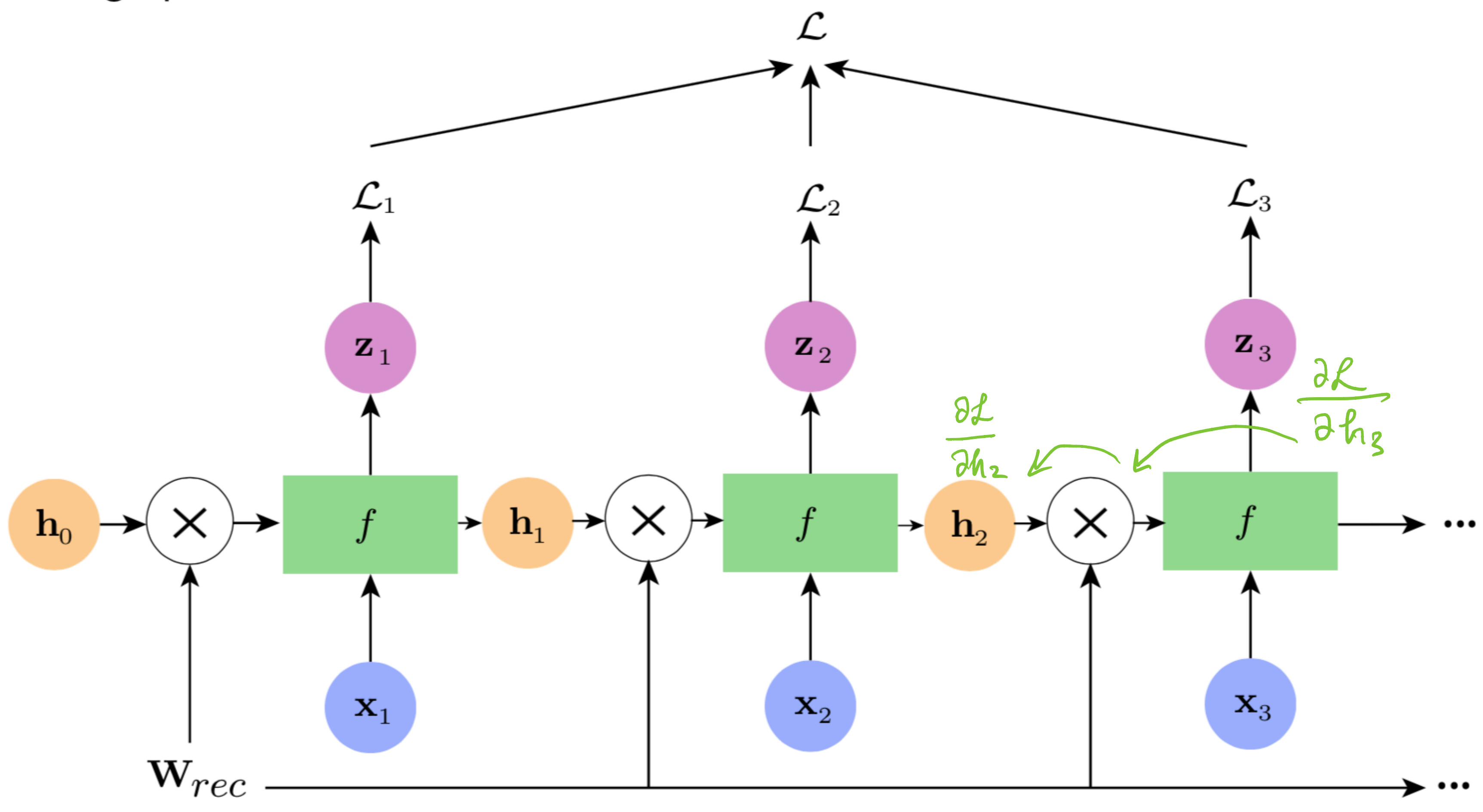


How does gradient flow in this graph?

$$\frac{\partial \mathcal{L}}{\partial h_{t-1}} = W_{rec}^T \, \mathbb{I}(W_{rec} h_{t-1} > 0) \odot \frac{\partial \mathcal{L}}{\partial h_t}$$

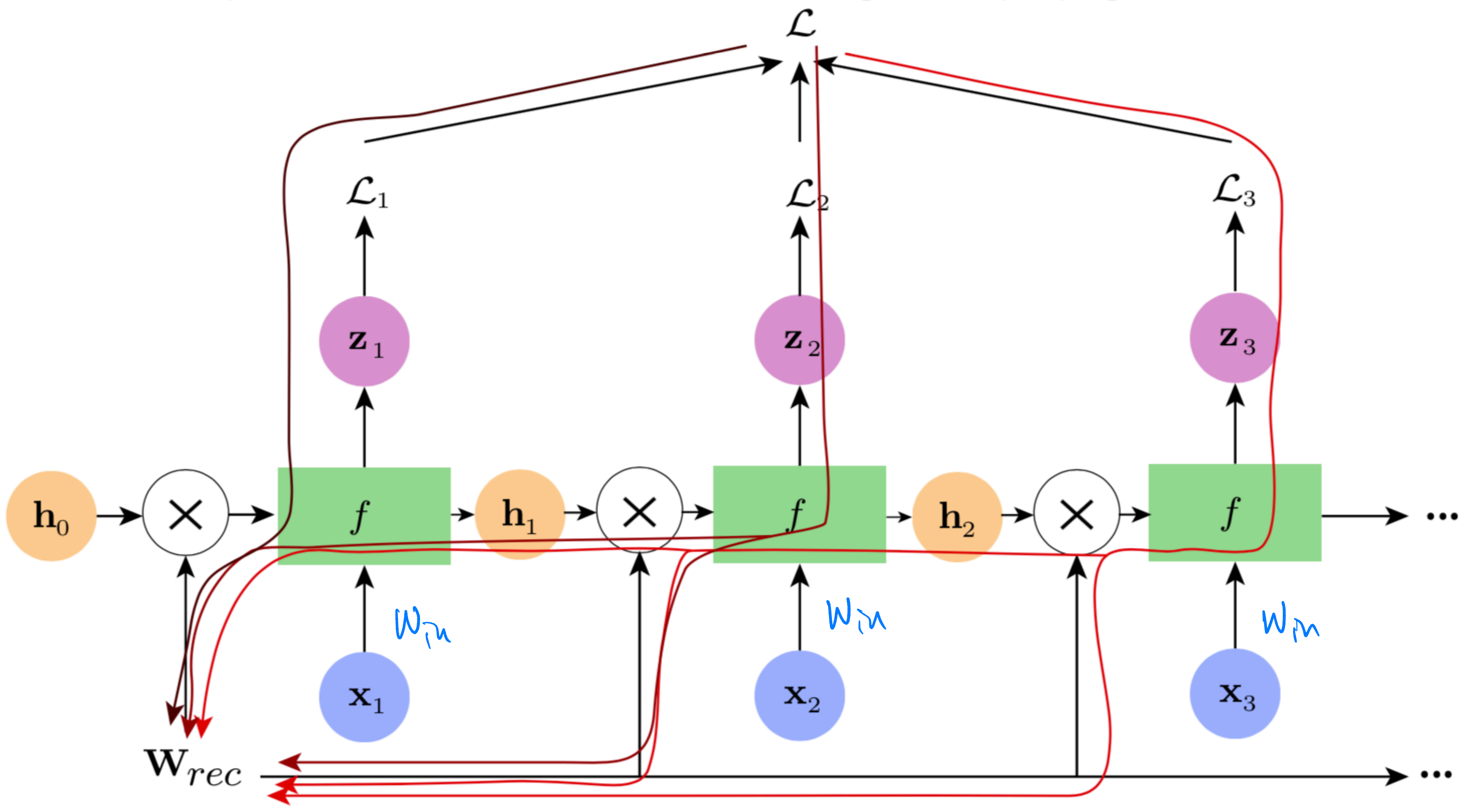
RNN training (cont.)

This graph can be redrawn as follows:



RNN training (cont.)

Redrawing the graph in this way, we see that to do backpropagation, there are several gradient paths to the parameters. Here, simply consider $\mathbf{W}_{\text{rec}}$. The red lines are all paths from the loss to $\mathbf{W}_{\text{rec}}$ through backpropagation.



Vanishing and exploding gradients

Vanishing and exploding gradients

There are a few important considerations then for backpropagation through time. The first of these is vanishing and exploding gradients.

As the number of layers to backpropagate through is the length of your input sequence in time, training these networks is like training a deep network with a bit of gradient injected at every time step (through $\mathcal{L}_t$). With RNNs, the problem can be more precisely formulated.

Every backpropagation step from $\mathbf{h}_{t+1}$ to $\mathbf{h}_t$ will require backpropagating through the nonlinearity f, and then a matrix multiply with $\mathbf{W}_{\text{rec}}$. If our nonlinearity is ReLU and our upstream gradient is $\partial \mathcal{L}_t / \partial \mathbf{h}_{t-k}$, then backpropagating to $\mathbf{h}_{t-k-1}$ involves the following computation:

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{h}_{t-k-1}} = \mathbf{W}_{\text{rec}}^T \left(\mathbb{I}(\mathbf{h}_{t-k-1} \geq 0) \odot \partial \mathcal{L}_t / \partial \mathbf{h}_{t-k} \right)$$

And hence going back Δt layers in time requires repeated multiplication by $\mathbf{W}_{\text{rec}}^T$ (Δt times).

Vanishing and exploding gradients (cont.)

Let $\mathbf{W}_{\text{rec}}^T$ have eigenvalue decomposition $\mathbf{U}\Lambda\mathbf{U}^{-1}$. Then multiplication by by this matrix Δt times is equivalent to:

$\Delta t = 100$

$$\begin{aligned}\left(\mathbf{W}_{\text{rec}}^T\right)^{\Delta t} &= \left(\mathbf{U}\Lambda\mathbf{U}^{-1}\right)^{\Delta t} \\ &= \mathbf{U}\Lambda\cancel{\mathbf{U}^{-1}\mathbf{U}}\Lambda\mathbf{U}^{-1}\cdots \\ &= \mathbf{U}\Lambda^{\Delta t}\mathbf{U}^{-1}\end{aligned}$$

$\lambda_1 = 1.1 \quad \lambda_1^{100} = 13780$

$\lambda_1 = 0.9 \quad \lambda_1^{100} = 2.65\times 10^{-5}$

Let λ_i be the ith eigenvalue of Λ. Then, along eigenvectors where $\lambda_i < 1$, the gradients will be attenuated to zero (vanishing gradients). Along dimensions where $\lambda_i > 1$, the gradients will grow exponentially (exploding gradients). These cause problems for gradient descent at earlier layers.

$$\Lambda = \begin{bmatrix} \lambda_1 & & & 0 \\ & \lambda_2 & & \\ & & \ddots & \\ 0 & & & \lambda_n \end{bmatrix} \qquad \Lambda^{100} = \begin{bmatrix} \lambda_1^{100} & & & 0 \\ & \lambda_2^{100} & & \\ & & \ddots & \\ 0 & & & \lambda_n^{100} \end{bmatrix}$$

Vanishing and exploding gradients

Addressing vanishing and exploding gradients

There are a few ways to address the problem of vanishing and exploding gradients. But first, a question:

Consider the gradient injected by $\mathcal{L}_1$. Doesn't this help to train the weights $\mathbf{W}_{\text{rec}}$, and if so, why should I be concerned that gradients from $\mathcal{L}_t$ are inaccurate? (e.g., think of the gradient injected by $\mathcal{L}_1$ as analogous to the auxiliary classifiers of GoogLeNet. Why doesn't this solve the problem of inaccurate gradients at earlier time steps?)

Vanishing and exploding gradients

Addressing vanishing and exploding gradients (cont.)

A general way to address the vanishing and exploding gradients problem is to do *truncated* backpropagation through time. In truncated backpropagation through time, instead of backpropagating all gradients back to the activations at time 1, we only backpropagate them p layers through time.

$\Delta t = 20$ time steps

Vanishing and exploding gradients

Exploding gradients. Like in CNNs, we can address the exploding gradients problem by constraining their norm to be a certain value. Let's say that the maximum gradient norm is a constant c. Then if the gradient, g, is such that $\|\mathbf{g}\| \geq c$, then

$$\mathbf{g} \leftarrow \frac{c}{\|\mathbf{g}\|}\mathbf{g}$$

Vanishing and exploding gradients

Vanishing gradients. Usually the way to handle vanishing gradients is via changing the architecture (later on in this lecture). However, there are ways to regularize RNNs to help ameliorate the vanishing gradients problem. Pascanu and colleagues (2012) propose the following regularization, which is added to the loss function:

$$\Omega = \sum_t \left(\frac{\left\| \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t} \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|}{\left\| \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \right\|} - 1 \right)^2 \longrightarrow \frac{\partial \mathcal{L}}{\partial h_t}$$

$$\left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\| \approx \left\| \frac{\partial \mathcal{L}}{\partial h_{t+1}} \right\|$$

Weight initialization

Weight initialization strategies

In CNNs, we saw that the Xavier and He initializations can make a big difference in training neural networks. How should we initialize RNN's? The following strategies are for networks using ReLU's.

- Le et al., 2015, suggest an "initialization trick" of setting $\mathbf{W}_{\text{rec}} = \mathbf{I}$ and $\mathbf{b}_{\text{rec}} = \mathbf{0}$. The intuition for why this is good is that it preserves the gradients going back in time (at least to begin), i.e., its eigenvalues are all 1 so gradients at the start are not heavily attenuated or amplified.
- Talathi et al., 2016, hypothesize that an initialization where one eigenvalue is equal to 1 and the rest are less than 1 is better. Their initialization is as follows:
  - Sample a matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ whose values are drawn from $\mathcal{N}(0, 1)$, and N is the number of units in the RNN.
  - Compute $\mathbf{B} = \frac{1}{N}\mathbf{A}\mathbf{A}^T$ and let λ_{max} be the largest eigenvalue of the matrix $\mathbf{B} + \mathbf{I}$.
  - Initialize $\mathbf{W}_{\text{rec}} = \frac{1}{\lambda_{\text{max}}}\mathbf{B} + \mathbf{I}$.

  Empirically this is better than initializing $\mathbf{W}_{\text{rec}} = \mathbf{I}$.

LSTM

The long short-term memory

The long short-term memory (LSTM) is a particular RNN architecture that is well-suited for addressing the problem of vanishing and exploding gradients. It was proposed by Hochreiter and Schmidhuber in 1997. It is one of the most commonly used RNN architectures today.

The name refers to its ability to store *short-term* memory for a *long* period of time. Hopefully the next few slides will help unpack what this really means.

LSTM

$W_f = [W_{rec}^f \quad W_{in}^f]$

The standard LSTM is defined as follows:

$h_t \in \mathbb{R}^n$

$x_t \in \mathbb{R}^m$

$W_f \in \mathbb{R}^{n \times (n+m)}$

"gates"

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\left(\underline{\mathbf{W}_f} \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right) \\
\underline{\mathbf{i}_t} &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \\
\underline{\mathbf{v}_t} &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\
\underline{\mathbf{o}_t} &= \sigma\left(\mathbf{W}_o \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_o\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{v}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
$$

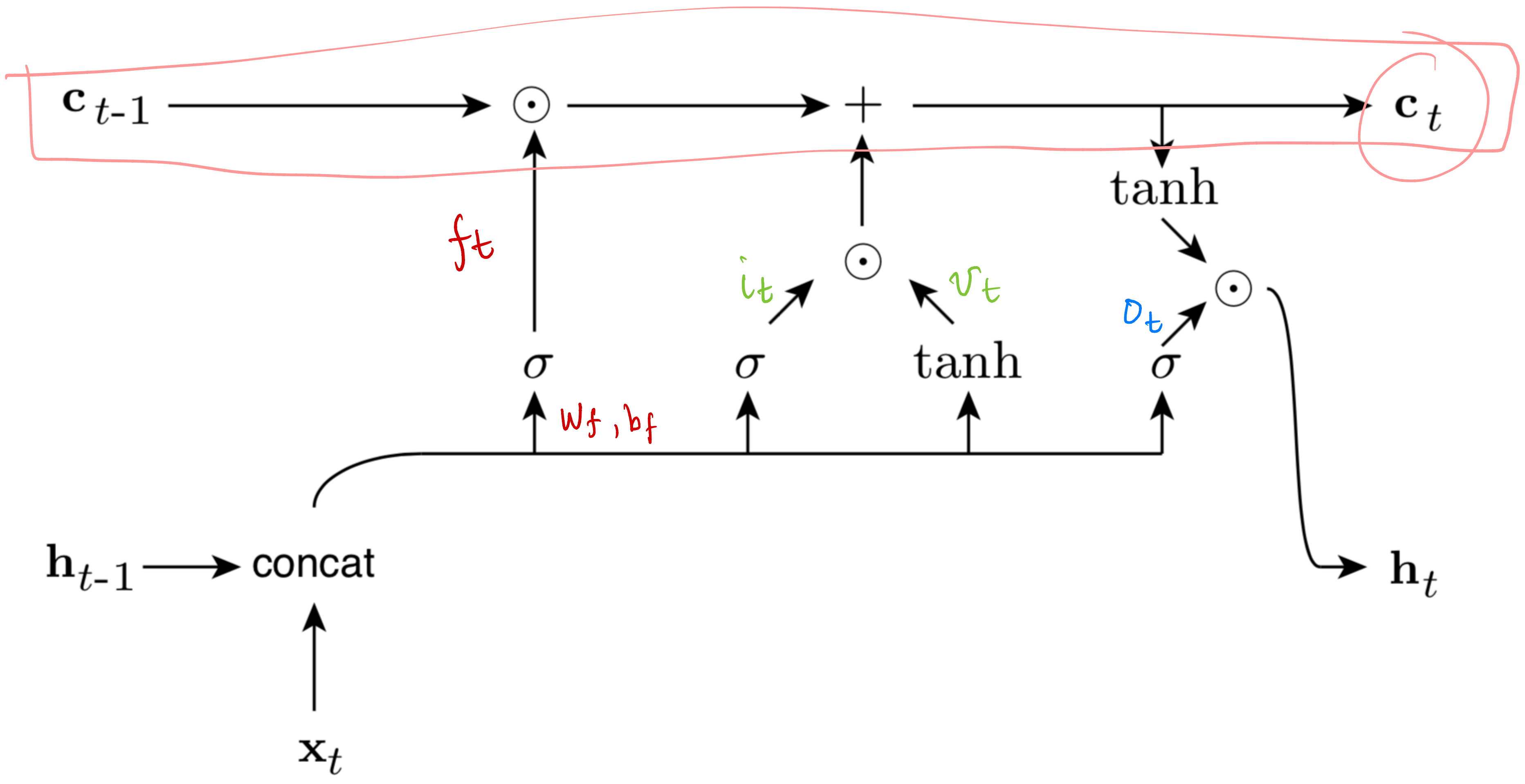
"values"

where $\sigma(\cdot)$ is the sigmoid function. These functions at first glance are opaque. Here, we will unpack what they mean and how they help solve the vanishing and exploding gradients problem.

LSTM

LSTM, block diagram

Here is a block diagram of the LSTM cell.



LSTM, cell state

The LSTM cell state, $\mathbf{c}_t$, is the central component of the LSTM. We think of the cell state as some memory or tape; it holds some value and remembers it. But the key thing is that we can alter this cell state (i.e., we can alter the memory or tape). At each point in time, there are three things we can do to the cell state:

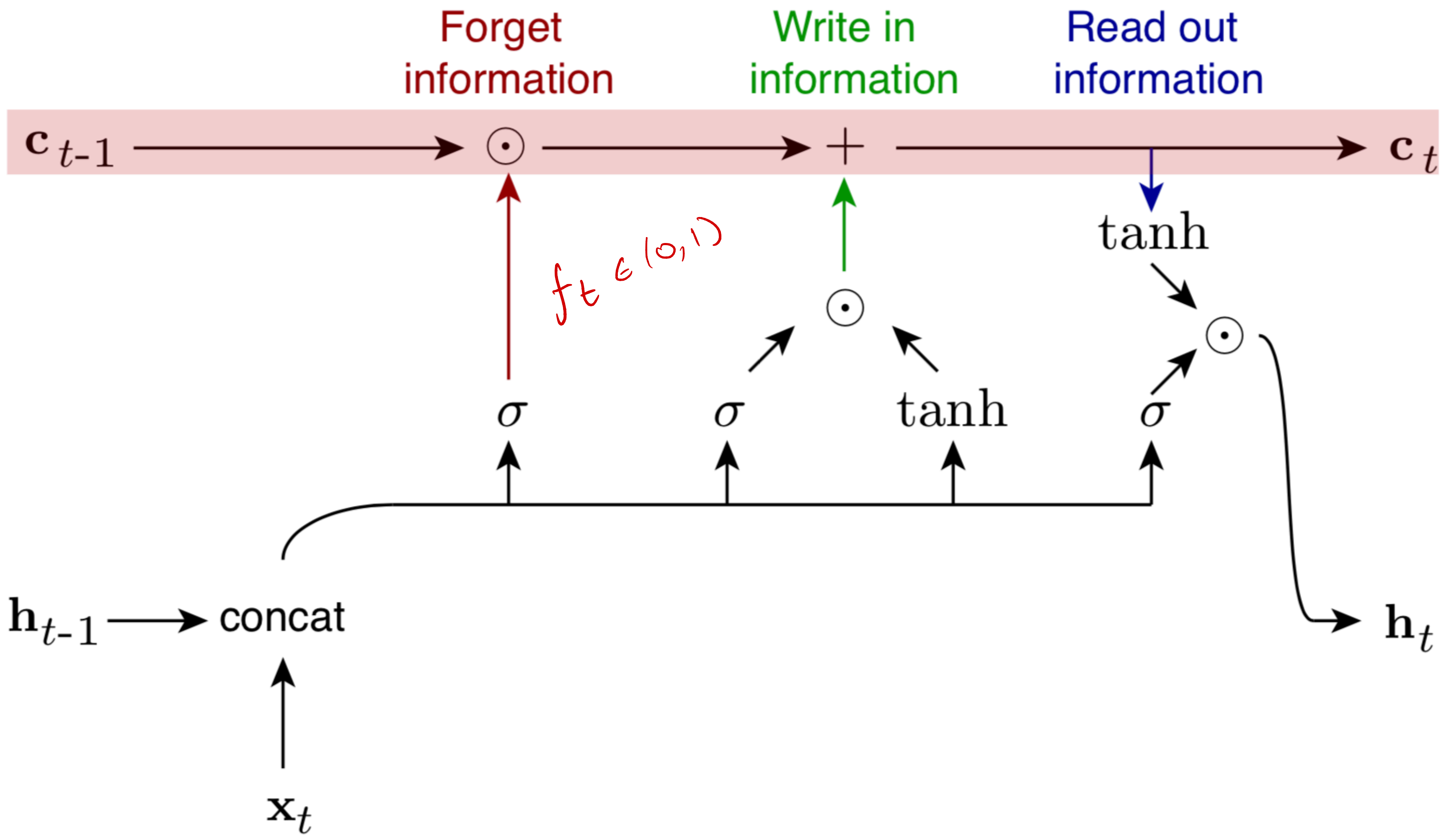
1. Forget information.
2. Write-in information.
3. Read-out information.

This is illustrated on the next page.

Note that the next hidden state, $\mathbf{h}_t$, is essentially a read out of the cell state $\mathbf{c}_t$, as $\mathbf{h}_t = f(\mathbf{c}_t)$, so the cell state contains all the vital information in the LSTM.

LSTM

LSTM, cell state



Forgetting information, writing in information, and reading out information are all mediated by gates.

LSTM

$f_t = \sigma(\text{affine})$

$\in (0, 1)$

LSTM, forgetting information

To forget information, the LSTM uses the forget gate, $\mathbf{f}_t$. This comprises the update, $\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t$. Note:

- If $\mathbf{f}_t$ is close to 1, then $\mathbf{c}_t \approx \mathbf{c}_{t-1}$. Thus, when the forget gate is large, most information is remembered.
- If $\mathbf{f}_t$ is close to 0, then $\mathbf{c}_t \approx 0$. Thus, when the forget gate is small, most information is forgotten.

This gate is illustrated on the next page.

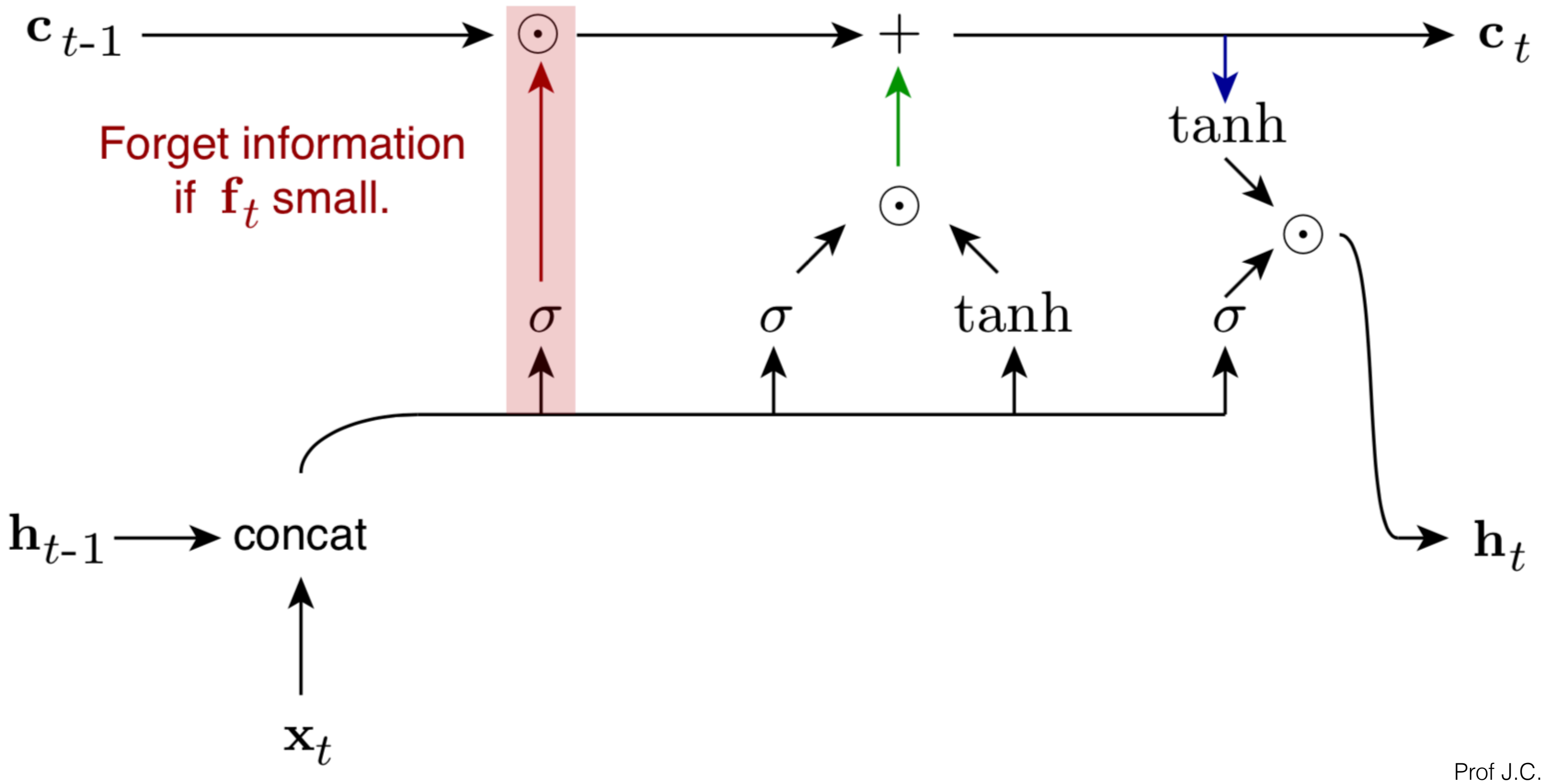
LSTM

LSTM, forget gate

The network computes a linear transformation

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_f\right)$$

and if $\mathbf{f}_t$ is small, it forgets the information. If $\mathbf{f}_t$ is large, it remembers the information.



LSTM

LSTM, writing in information

To write in information, the LSTM needs to compute two values. The following is not convention, but it helps me remember. We'll call these gates the value gate, $\mathbf{v}_t$, and the input gate, $\mathbf{i}_t$. These are calculated as:

g_t

$$\begin{aligned} \text{"value"} \leftarrow \quad \mathbf{v}_t &= \tanh\left(\mathbf{W}_v \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_v\right) \\ \text{"input gate"} \leftarrow \quad \mathbf{i}_t &= \sigma\left(\mathbf{W}_i \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_i\right) \quad \in (0,1) \end{aligned}$$

- The value gate, $\mathbf{v}_t \in (-1, 1)$ tells us the value we want to add to the cell state.
- The input gate, $\mathbf{i}_t \in (0, 1)$ tells us how much of the value gate, $\mathbf{v}_t$ to write to the cell state.

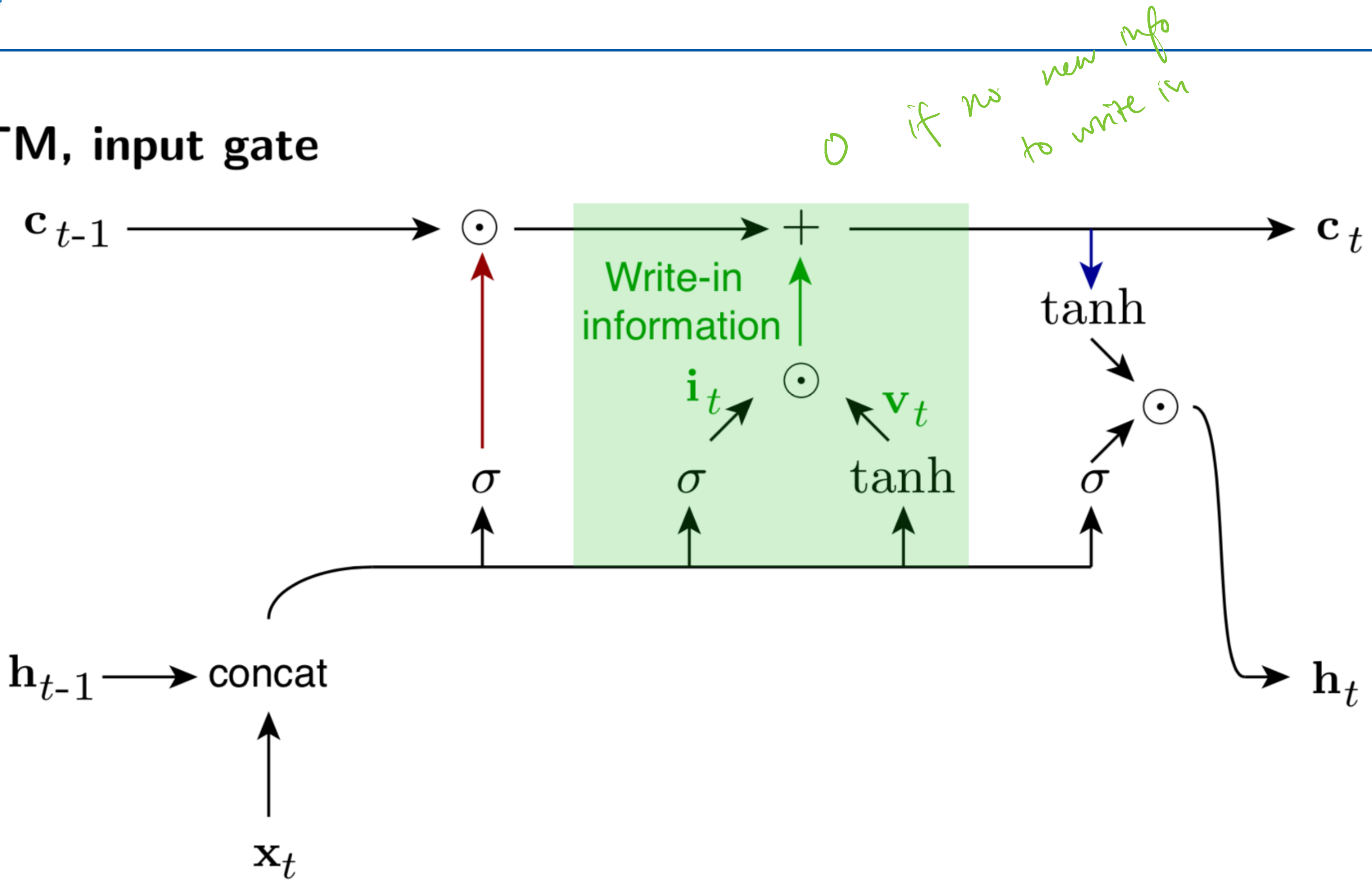
Then, the total information we get to write in to the cell state is the multiplication of these two, i.e., this update looks like:

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{v}_t$$

If either $\mathbf{v}_t$ or $\mathbf{i}_t$ is zero, we write no information to the cell state. If $\mathbf{i}_t = 1$, we write whatever update information $\mathbf{v}_t$ into the cell.

LSTM

LSTM, input gate



LSTM, reading out information

Now that we've updated the cell state by both forgetting information and writing in information, we are now ready to read out information to update the hidden state of the LSTM. This is fairly simple: we take our cell state, put it through the $\tanh$ nonlinearity, and then use our final gate, the output gate $\mathbf{o}_t$.

- The output gate, $\mathbf{o}_t$, tells us how much of the cell state is going to be read out to the hidden state.
- If $\mathbf{o}_t$ is small, very little will be read out.
- If $\mathbf{o}_t$ is large, a lot of the cell state will be read out.

Formally, the readout is:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

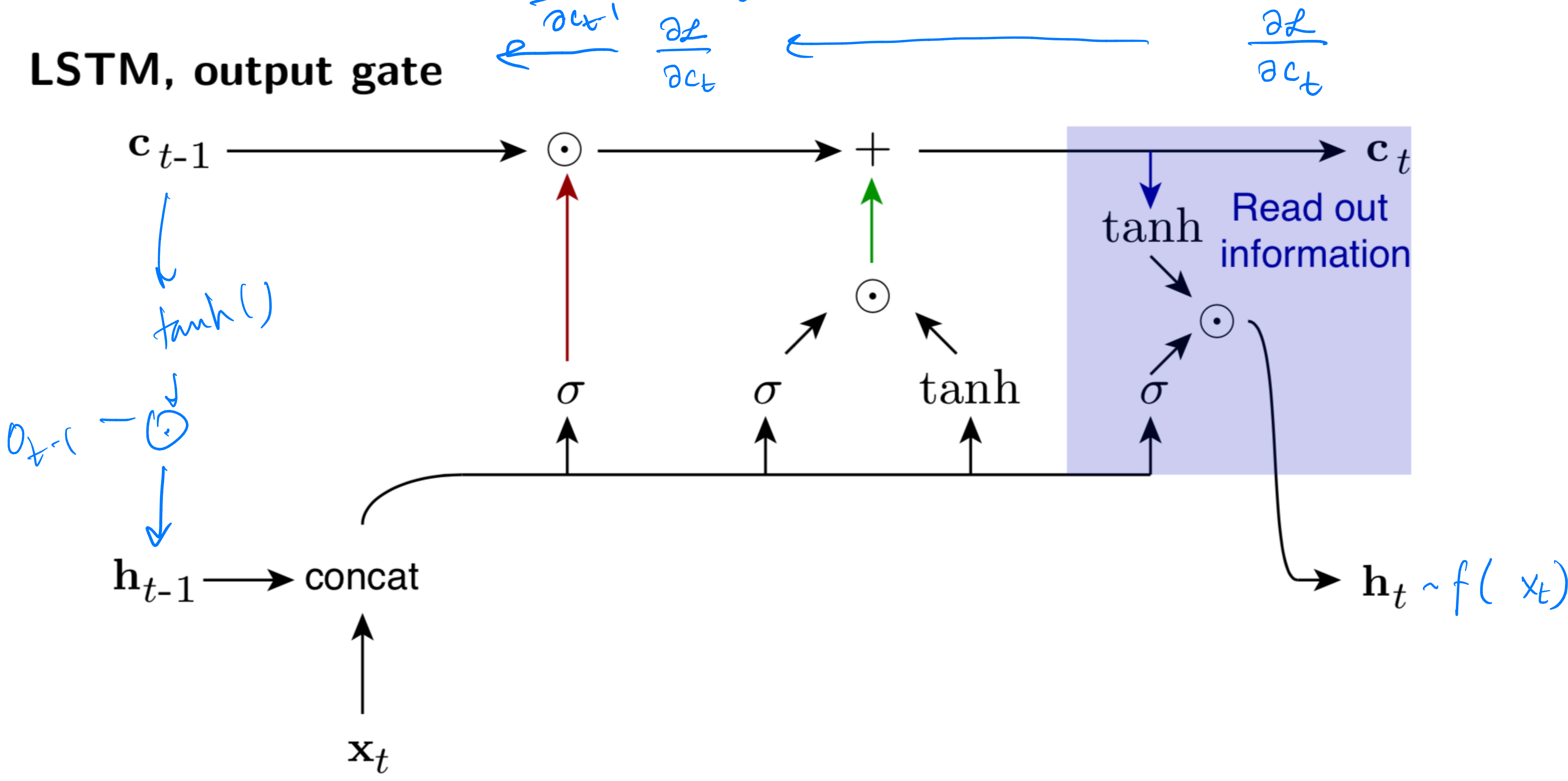
$$o_t = \sigma(\text{affine})$$

LSTM

if $f_t = 1$

$\frac{\partial \mathcal{L}}{\partial c_{t-1}} = \frac{\partial \mathcal{L}}{\partial c_t}$

LSTM, output gate



LSTM training

From this representation of the LSTM, we now can glean insight into why this works well.

- The cell state, $\mathbf{c}_t$, during backpropagation, has almost uninterrupted gradient flow. It's analogous to a gradient highway, much like in ResNets.
- In particular, the $+$ operation passes the gradient back.
- The gradient may be attenuated by the forget gate, $\mathbf{f}_t$. The gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} \odot \mathbf{f}_t$$

  and therefore if the forget gate is small, then $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t-1}}$ will be small, too.

LSTM, last comments

In practice, LSTMs are easier to train than vanilla RNNs using first order gradient descent techniques. However, note that the LSTM has (!) $4\times$ the number of parameters of a vanilla RNN.

To address this problem, another type of recurrent unit is used, called the **gated recurrent unit**.

GRU

Gated recurrent units (GRUs)

The standard GRU is defined as follows:

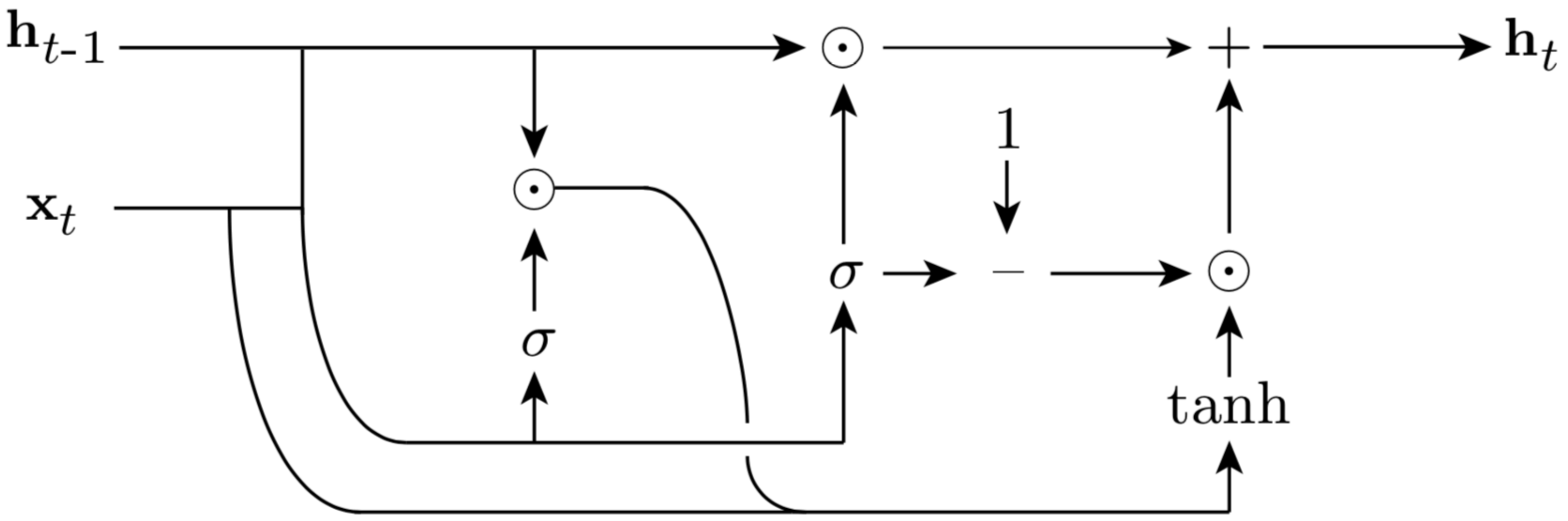
$$
\begin{aligned}
\mathbf{r}_t &= \sigma\left(\mathbf{W}_r \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_r\right) \\
\mathbf{u}_t &= \sigma\left(\mathbf{W}_u \begin{bmatrix} \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_u\right) \\
\tilde{\mathbf{h}}_t &= \tanh\left(\mathbf{W}_h \begin{bmatrix} \mathbf{r}_t \odot \mathbf{h}_{t-1} \\ \mathbf{x}_t \end{bmatrix} + \mathbf{b}_h\right) \\
\mathbf{h}_t &= \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

Again, it looks opaque at first, so let's unpack it. We call:

- $\mathbf{r}_t$ the reset gate.
- $\mathbf{u}_t$ the update gate.
- $\tilde{\mathbf{h}}_t$ the candidate activation.

GRU, block diagram

The GRU has the following block diagram:



Like the LSTM, we see that the hidden state essentially has a gradient highway for backpropagation.

GRU

GRU, update gate

The first thing we note about the GRU is that it has no cell state. So let's start with the update gate, $\mathbf{u}_t$. The update gate tells us how the hidden state updates.

- If the update gate $\mathbf{u}_t$ is close to 1, then the hidden state stays approximately constant: $\mathbf{h}_t \approx \mathbf{h}_{t-1}$.
- If the update gate $\mathbf{u}_t$ is close to 0, then the hidden state forgets its previous value and adopts the candidate activation, $\tilde{\mathbf{h}}_t$.

Hence, the GRU colloquially uses the hidden state as the "memory" as opposed to instantiating a new cell state. We can think of $\mathbf{u}_t$ being close to 1 as information not being forgotten, and $\mathbf{u}_t$ being close to 0 as information being forgotten and new information (i.e., the candidate activation) being written in.

GRU

GRU, reset gate

At this point, we're left with only one thing left to understand about the GRU – what is the new value $\tilde{\mathbf{h}}_t$ that may be written into the hidden state (if the update gate $\mathbf{u}_t$ is close to 0)?

This is mediated by the reset gate.

- If the reset gate $\mathbf{r}_t$ is close to 1, then the value $\tilde{\mathbf{h}}_t$ looks like a standard vanilla RNN update.
- If the reset gate $\mathbf{r}_t$ is close to 0, then the value $\tilde{\mathbf{h}}_t$ is completely set by the input.

The intuition is that if the reset gate $\mathbf{r}_t$ is close to 0, then it'll be as if the GRU was being reset, only looking at the input.

GRU

GRU, gradient flow

Much like the LSTM, the GRU is able to learn long term dependencies since there's a gradient highway on the hidden states $\mathbf{h}_t$. When it needs to remember long-term sequences, then the update gate will be close to (1), allowing the gradient to flow back in time through $\mathbf{h}_t$.

GRU, other comments

In this formulation of the GRU, it has $3\times$ the parameters of a vanilla RNN, which is less than the LSTM. (There is also a minimal GRU, which only has $2\times$ the parameters, and operates similarly to the GRU.) Compared to the LSTM:

- The GRU uses less memory and computation than the LSTM, since it doesn't maintain a cell state.
- GRUs have been empirically observed to train faster than LSTMs.
- LSTMs ought to be able to remember longer sequences by using a dedicated cell state.
- GRUs empirically tend to perform better than LSTMs, but this isn't always the case. In particular, in your application, it could be that a GRU does better than an LSTM, or vice versa. My recommendation is that you try both.

Training RNNs

A final note on training RNNs

Since RNNs tend to have less units, second order methods may be more plausible. For example, Martens, Sutskever and Hinton (2011) reported a "Hessian-free" conjugate gradient method to optimize RNNs, and even a more expressive architecture called a multiplicative RNN. Here, second order methods helped substantially.

Training RNNs

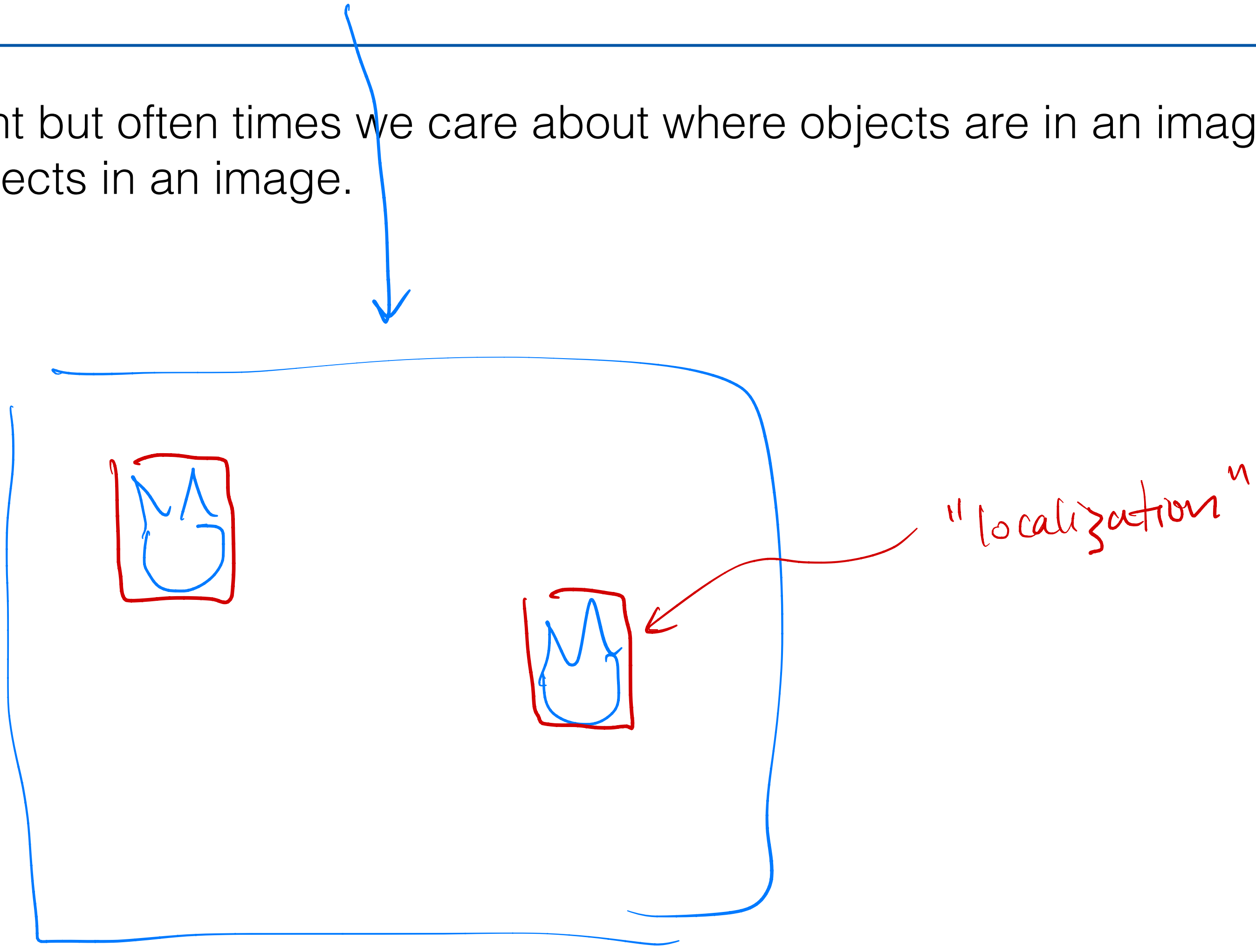
How to train?

In total, we have suggested four ways you might go about training recurrent neural networks.

- Use a vanilla RNN with gradient clipping (to ameliorate exploding gradients) and Pascanu's suggested regularization (to ameliorate vanishing gradients).
- Use an LSTM.
- Use a GRU.
- (Extension of 2 and 3: use some other gating unit variant (there are a bunch).)
- Consider using second order optimization methods.

Detection and Segmentation

Classification is important but often times we care about where objects are in an image, or to identify multiple objects in an image.



Pascal VOC dataset

Used in YOLO (to be described).

This dataset had a competition until 2012. There are **twenty classes**. They had both a classification and detection competition.

Classes:

- Person: person
- Animal: bird, cat, cow, dog, horse, sheep
- Vehicle: aeroplane, bicycle, boat, bus, car, motorbike, train
- Indoor: bottle, chair, dining table, potted plant, sofa, tv/monitor

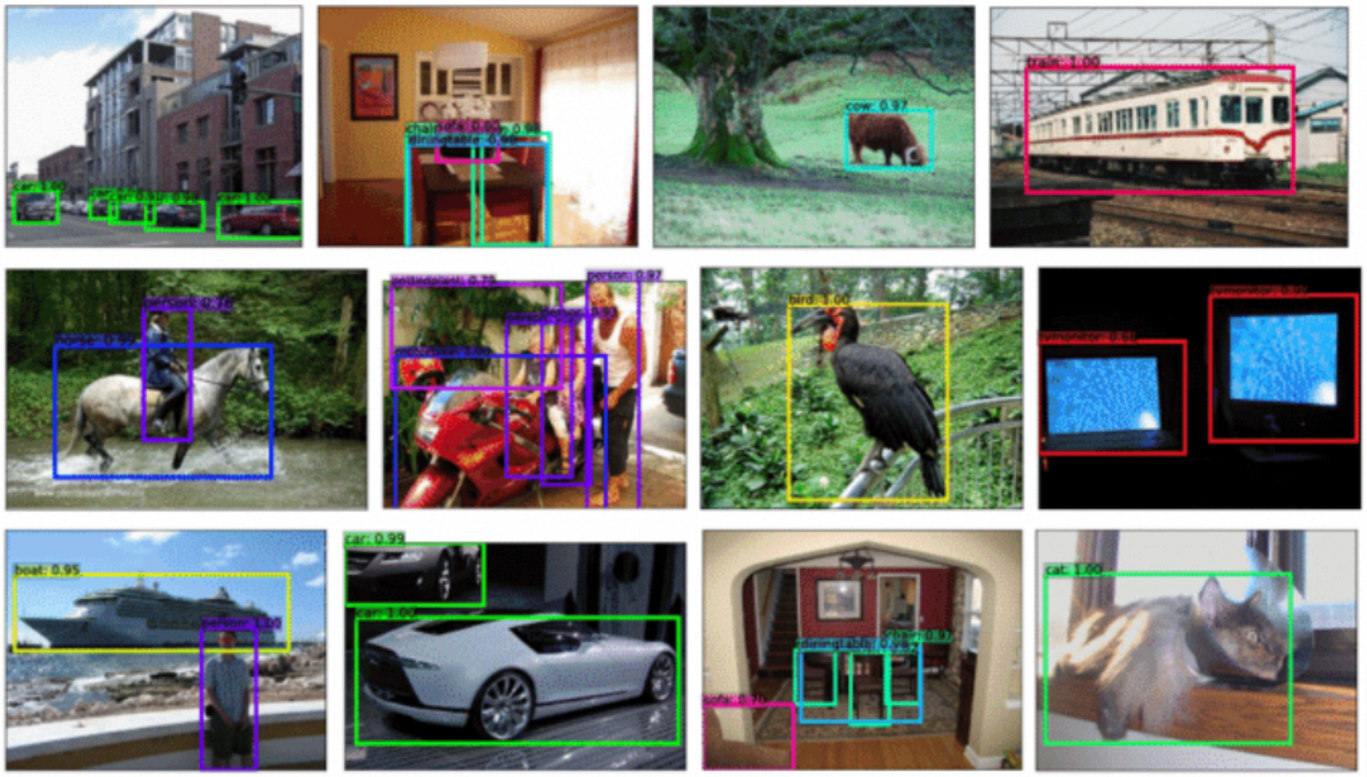
Classification and Detection Tasks:

- **Classification**: For each of the twenty classes, predicting presence/absence of an example of that class in the test image.

- **Detection**: Predicting the bounding box and label of each object from the twenty target classes in the test image.

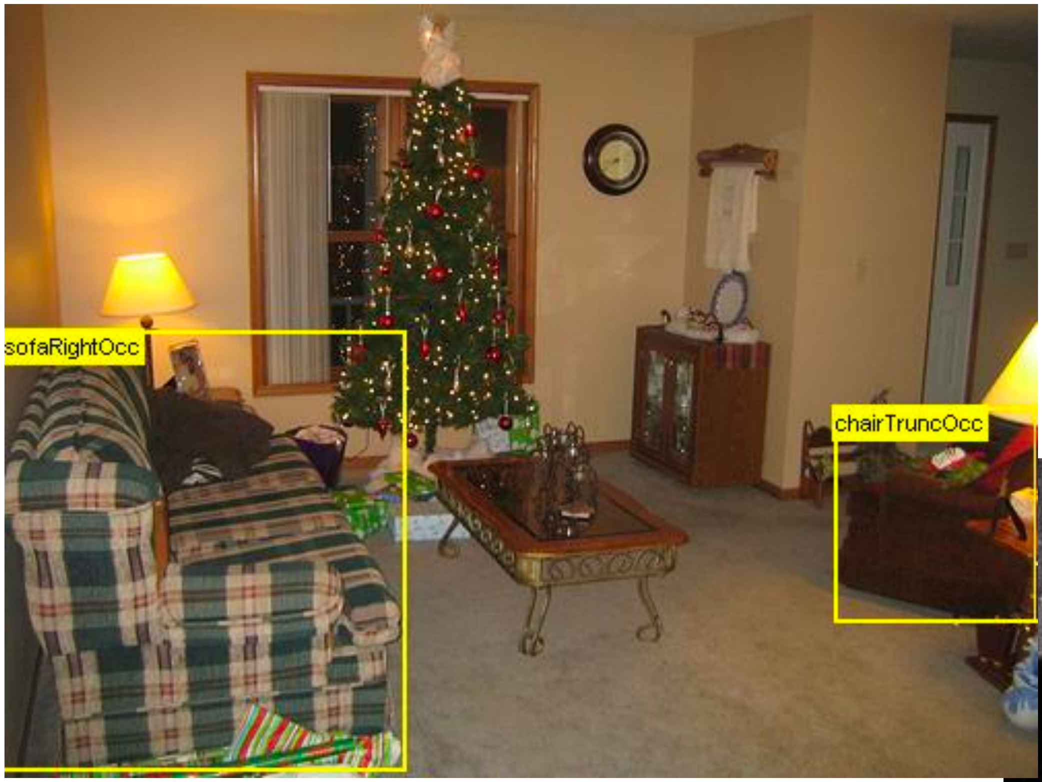
http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

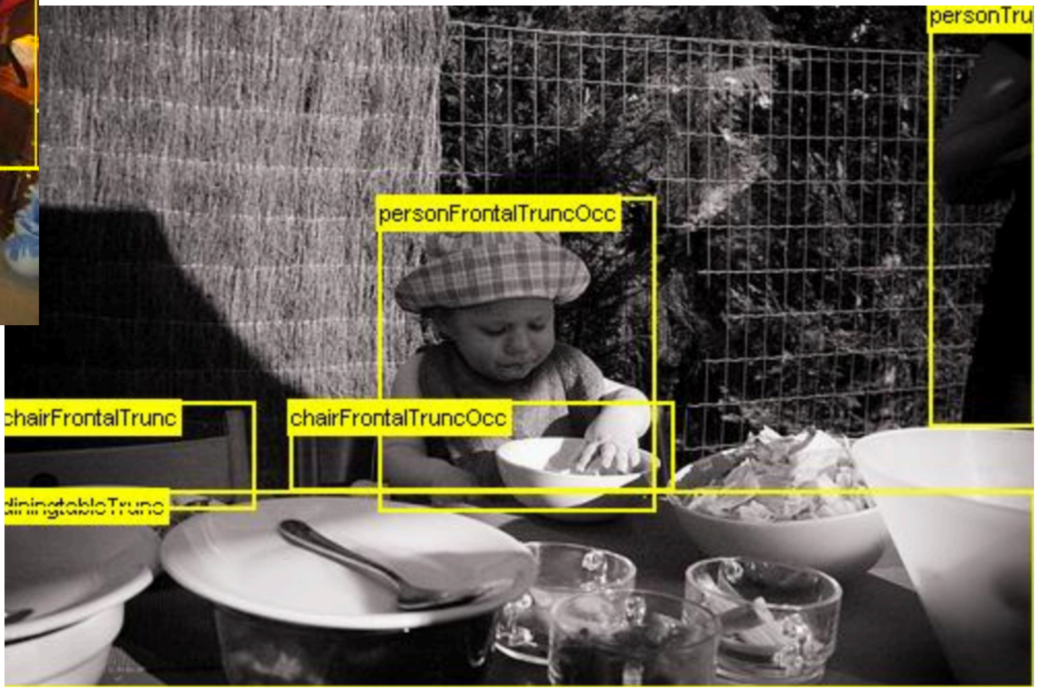
Pascal VOC dataset



http://host.robots.ox.ac.uk/pascal/VOC/voc2007/

Pascal VOC dataset



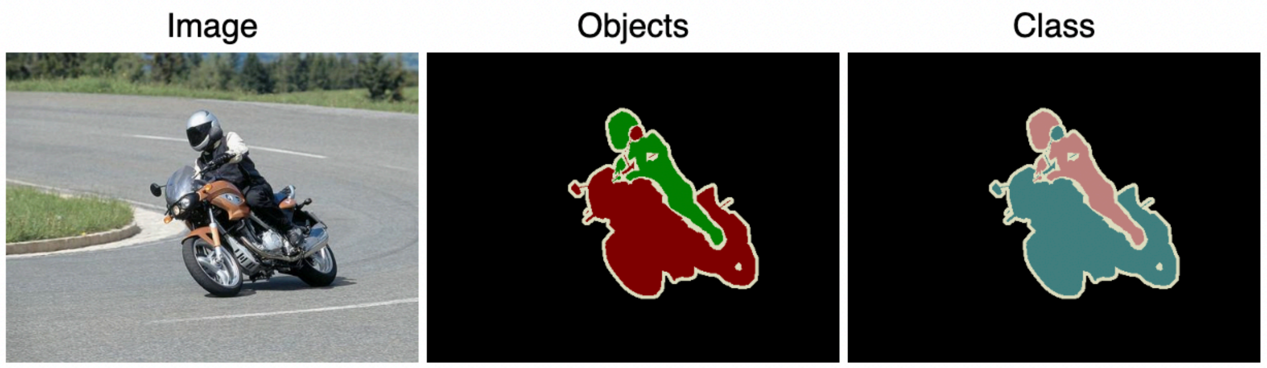


http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

Pascal VOC dataset

Segmentation:

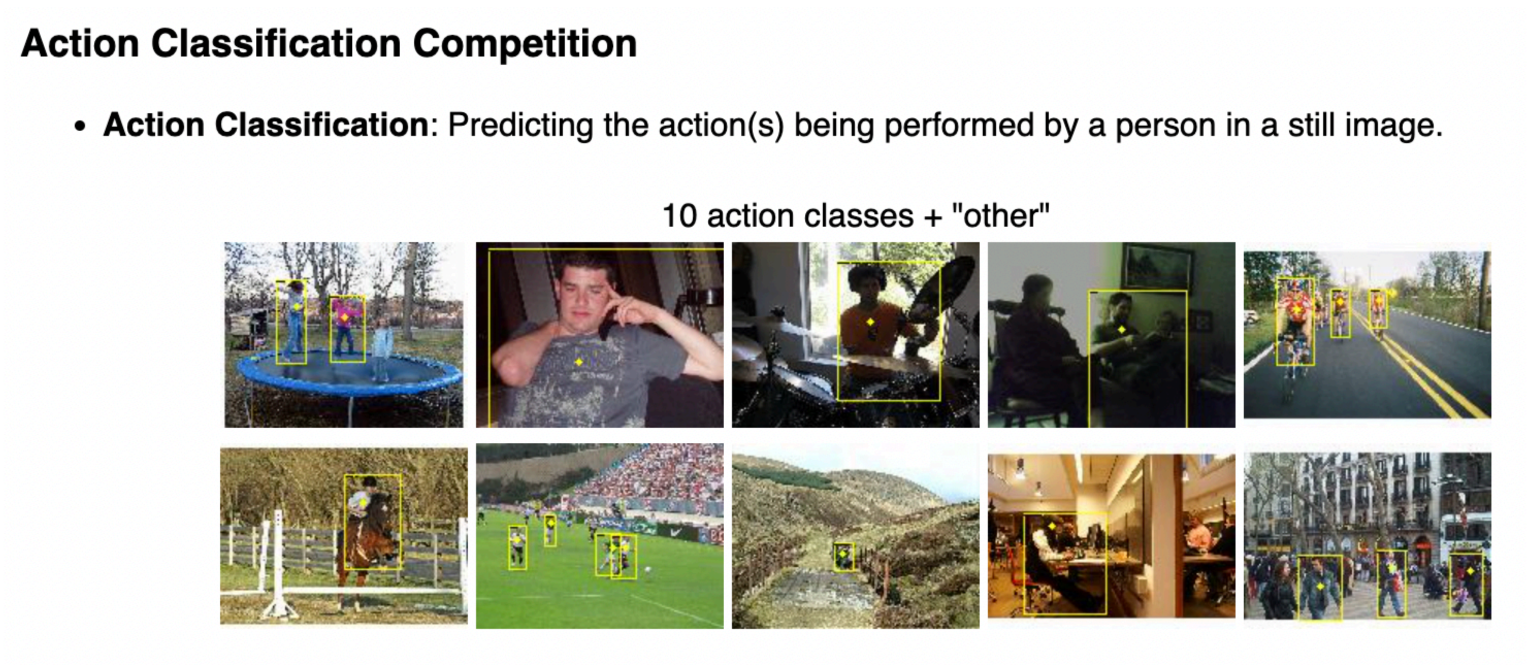
- **Segmentation**: Generating pixel-wise segmentations giving the class of the object visible at each pixel, or "background" otherwise.



http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

Pascal VOC dataset

Action detection:



Action Classification Competition

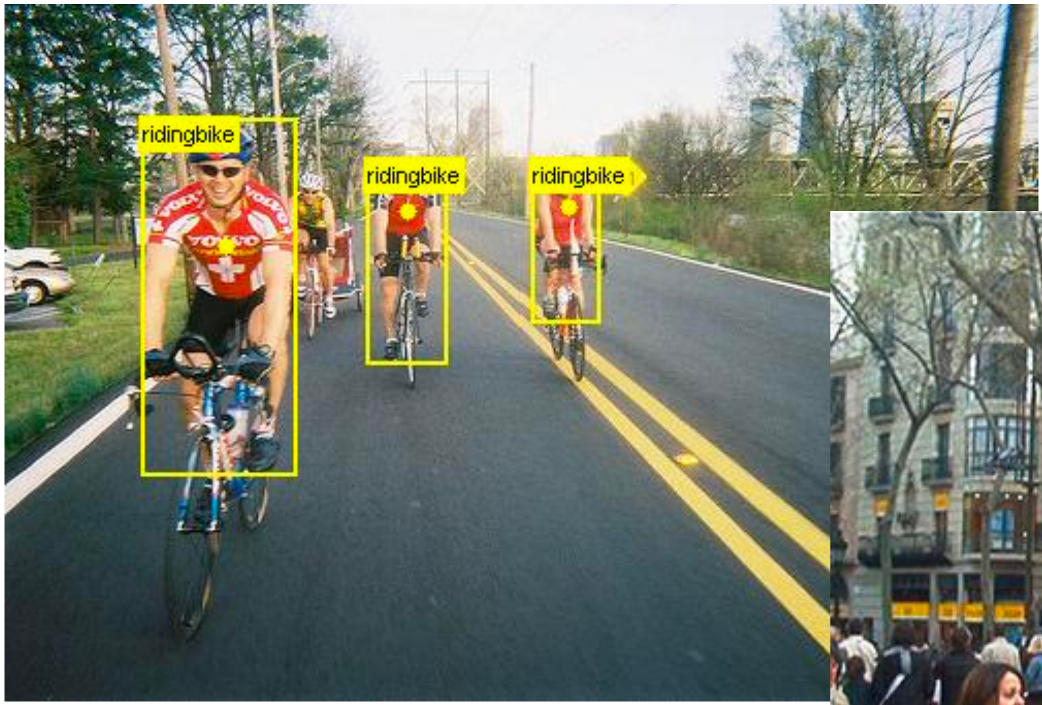
- **Action Classification**: Predicting the action(s) being performed by a person in a still image.

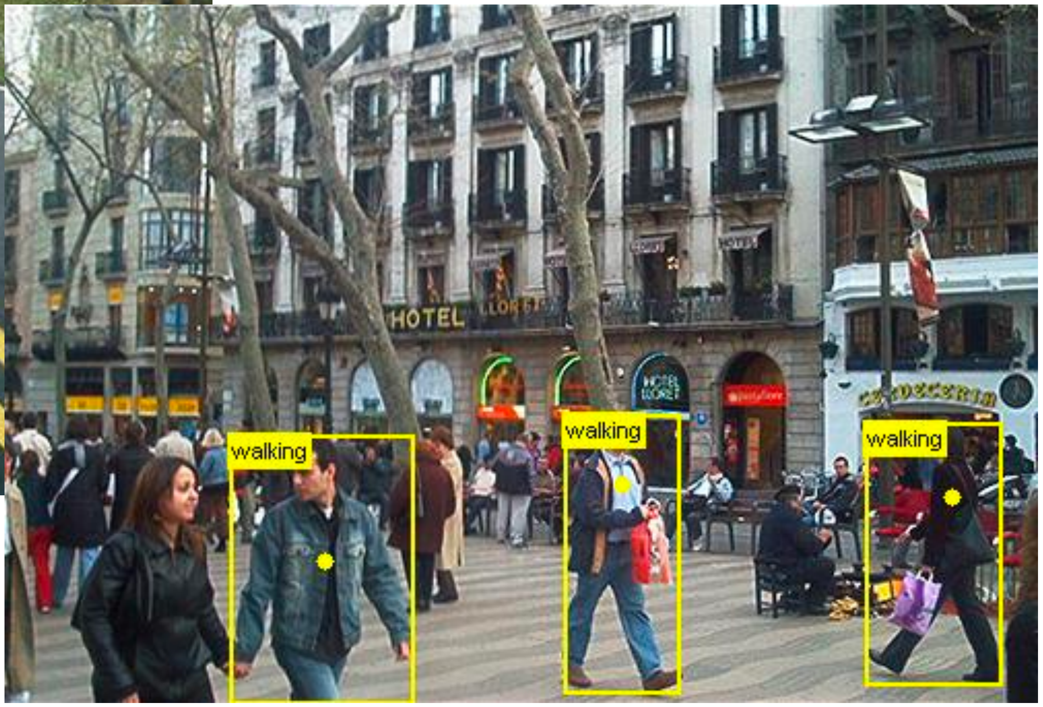
10 action classes + "other"

http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

Pascal VOC dataset

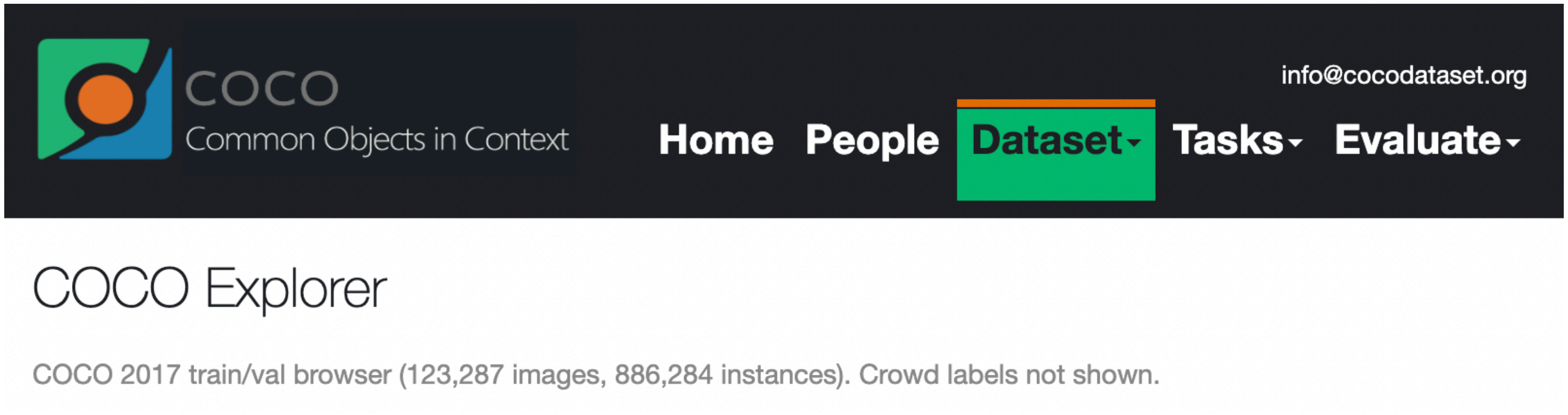
Action detection:





http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

COCO: Common Objects in Context

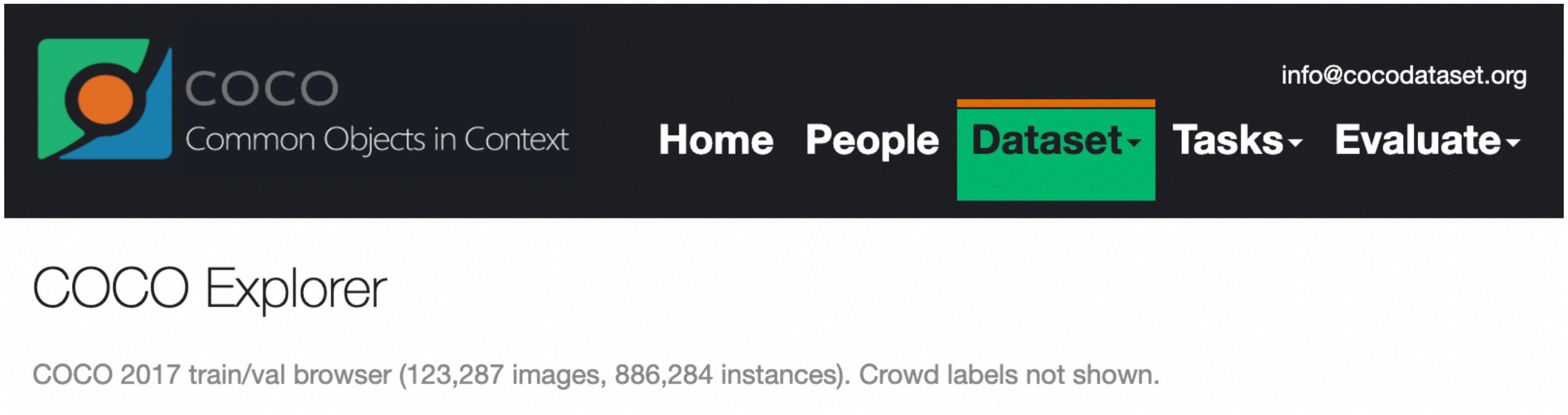


COCO Explorer

COCO 2017 train/val browser (123,287 images, 886,284 instances). Crowd labels not shown.

https://cocodataset.org/#home

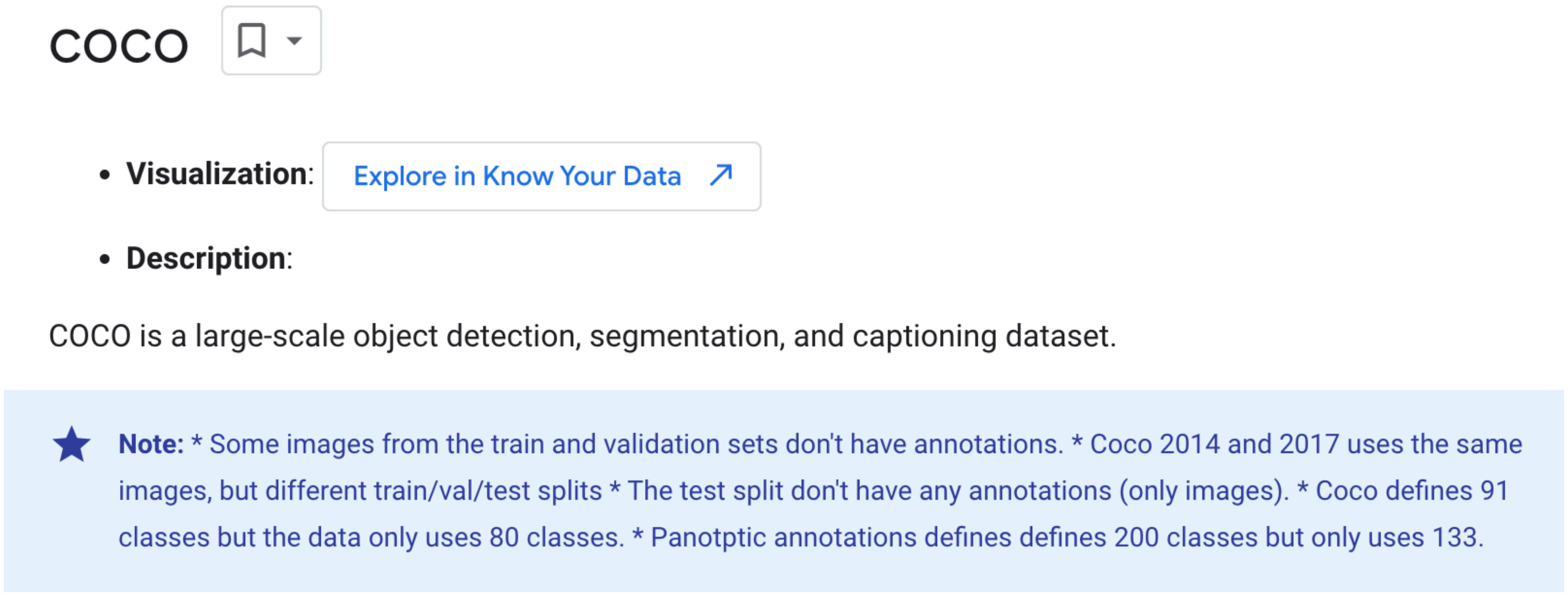
COCO: Common Objects in Context



COCO Explorer

COCO 2017 train/val browser (123,287 images, 886,284 instances). Crowd labels not shown.

https://cocodataset.org/#home



COCO

- **Visualization**: Explore in Know Your Data
- **Description**:

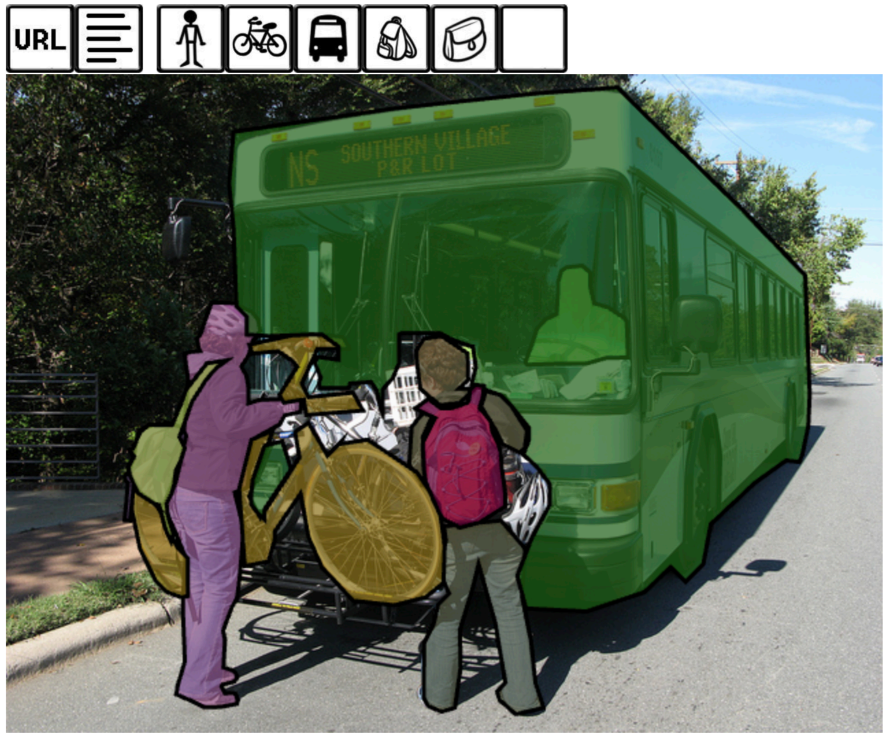
COCO is a large-scale object detection, segmentation, and captioning dataset.

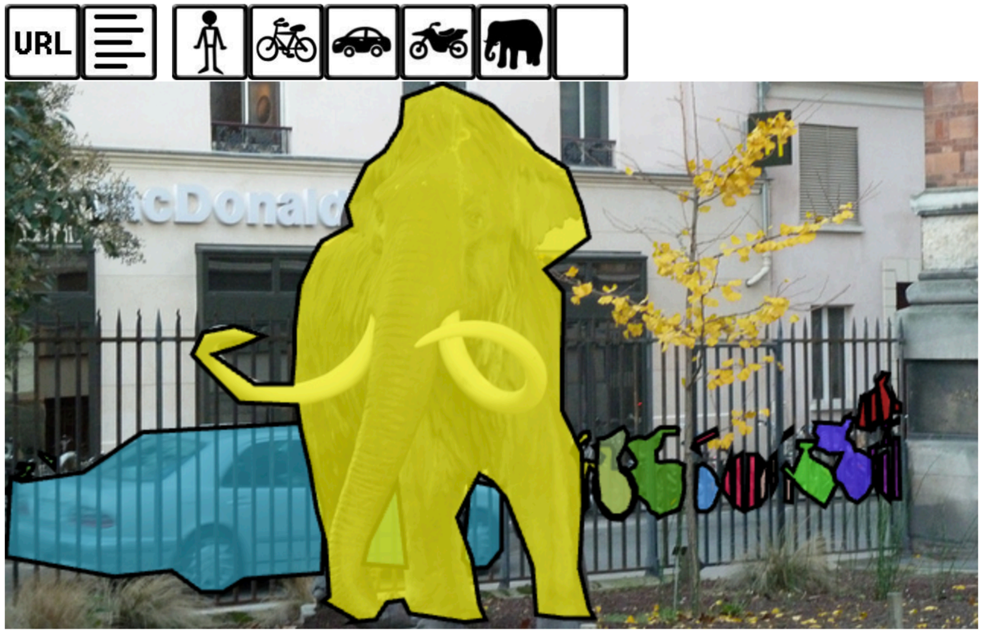
Note: * Some images from the train and validation sets don't have annotations. * Coco 2014 and 2017 uses the same images, but different train/val/test splits * The test split don't have any annotations (only images). * Coco defines 91 classes but the data only uses 80 classes. * Panotptic annotations defines defines 200 classes but only uses 133.

https://www.tensorflow.org/datasets/catalog/coco

COCO: Common Objects in Context

Search for bicycles:

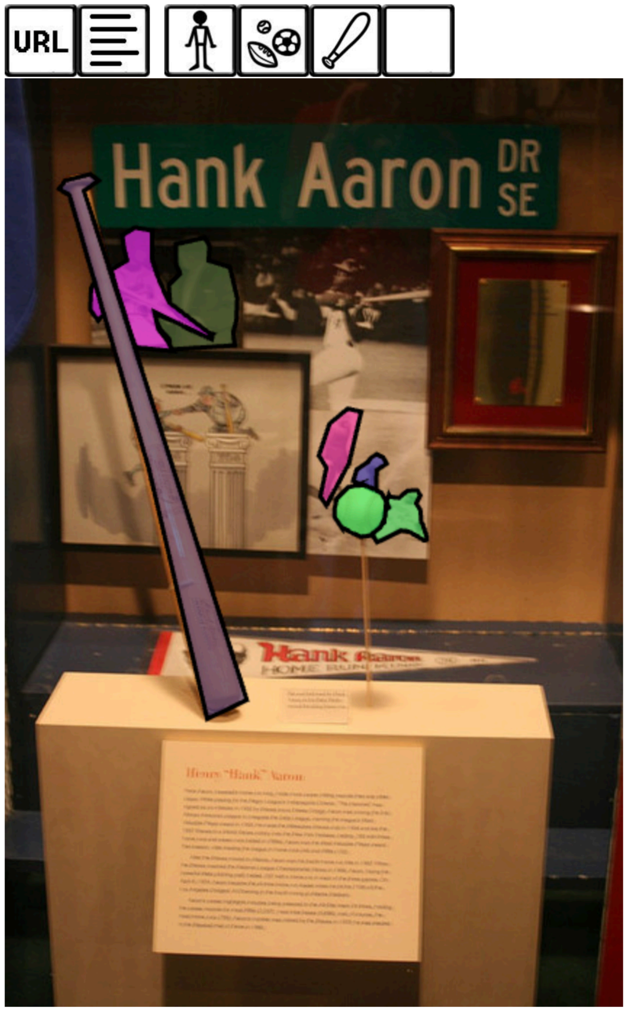


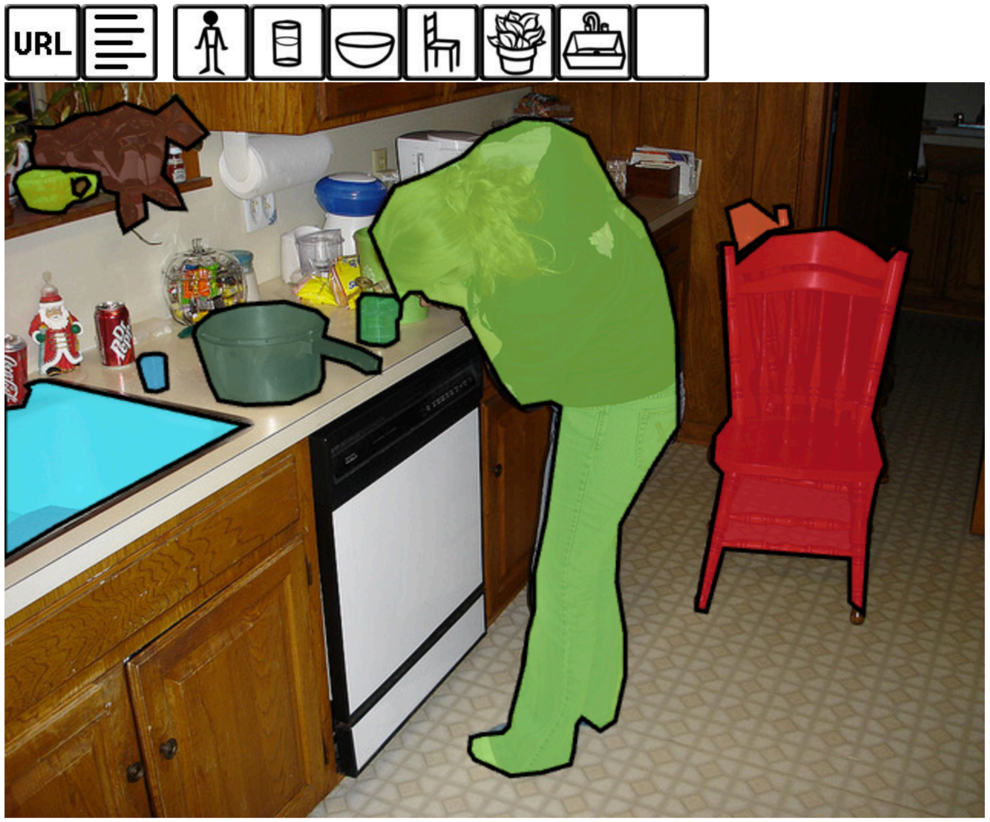


https://cocodataset.org/#home

COCO: Common Objects in Context

Search for persons:

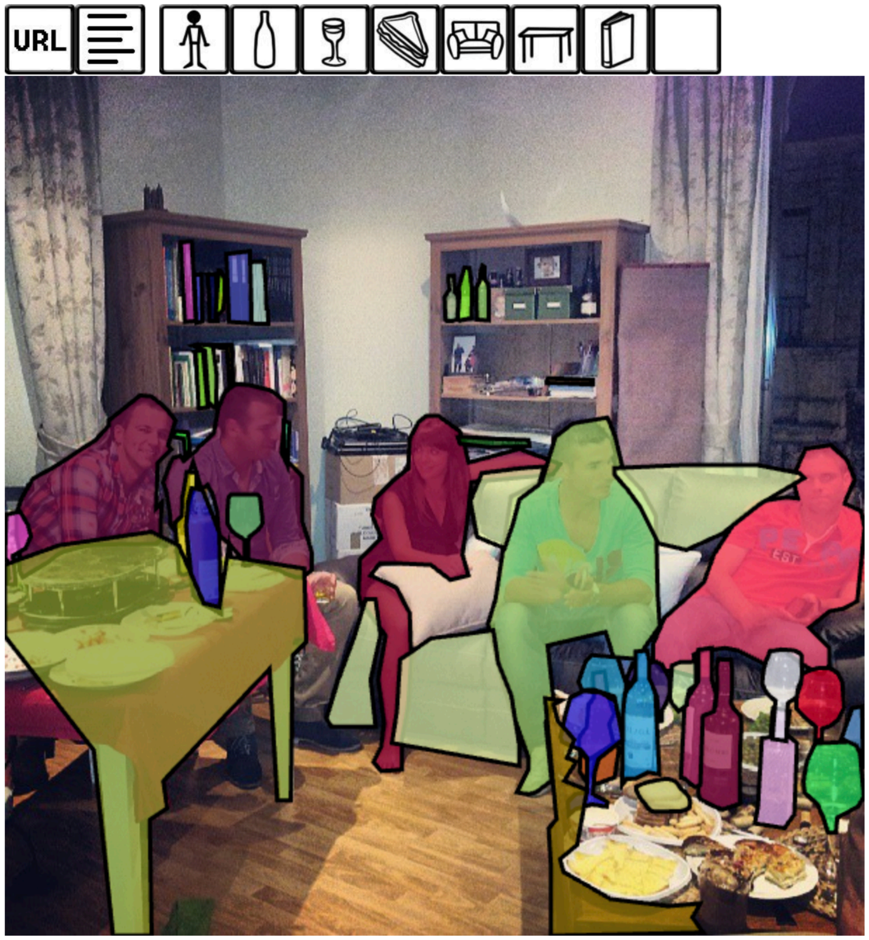


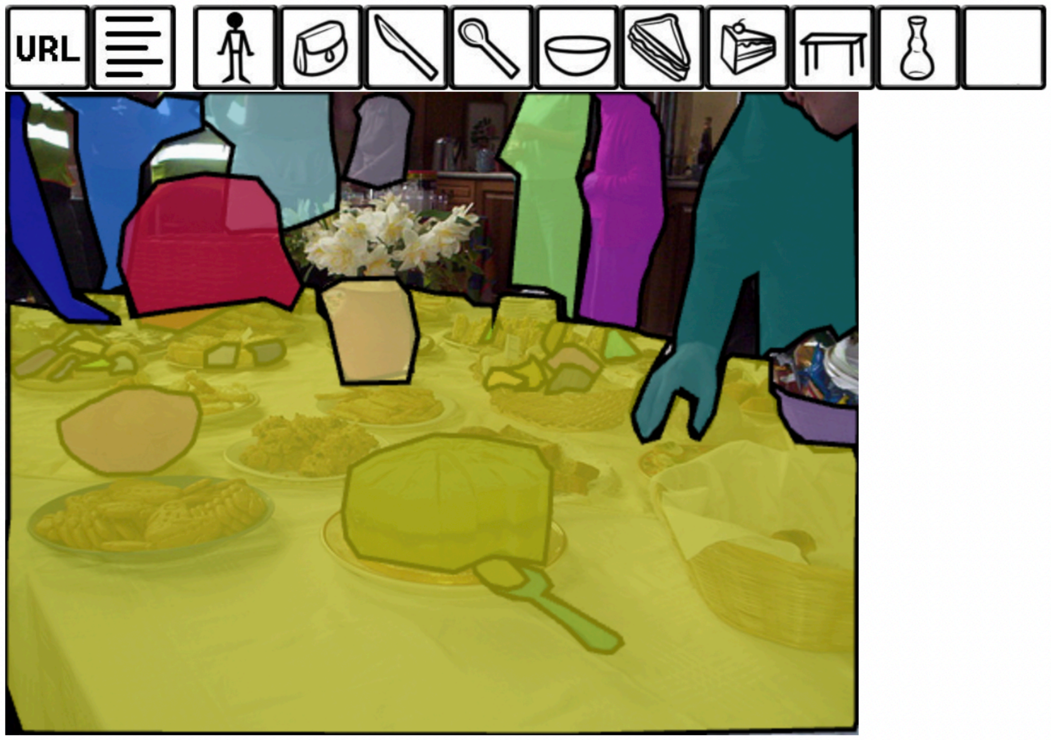


https://cocodataset.org/#home

COCO: Common Objects in Context

Search for persons + sandwiches:

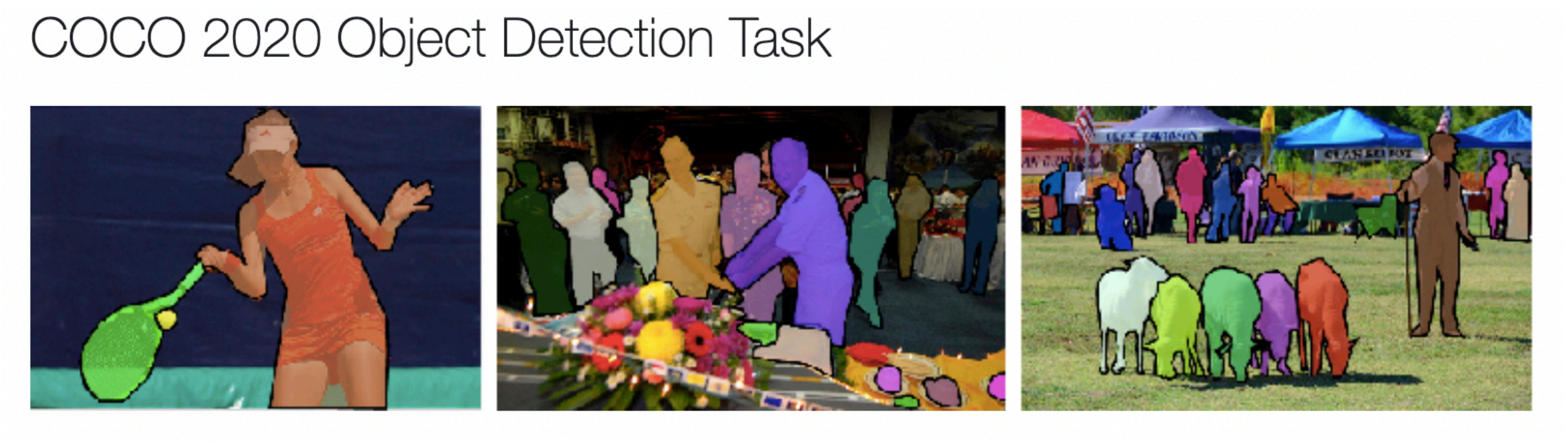




https://cocodataset.org/#home

COCO: Common Objects in Context

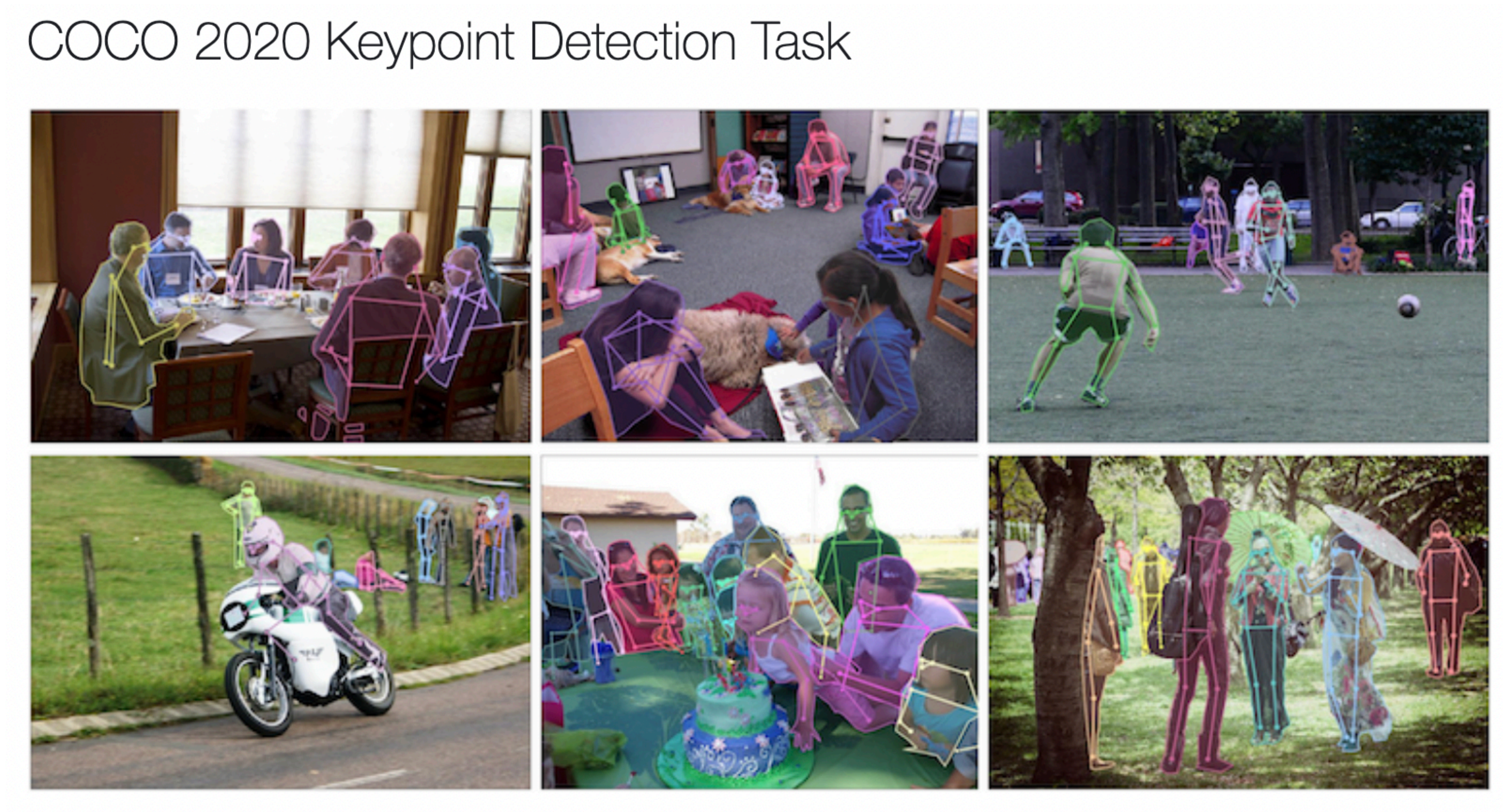
Tasks (competitions until 2020):


COCO 2020 Object Detection Task

https://cocodataset.org/#home

COCO: Common Objects in Context

Tasks (competitions until 2020):


COCO 2020 Keypoint Detection Task

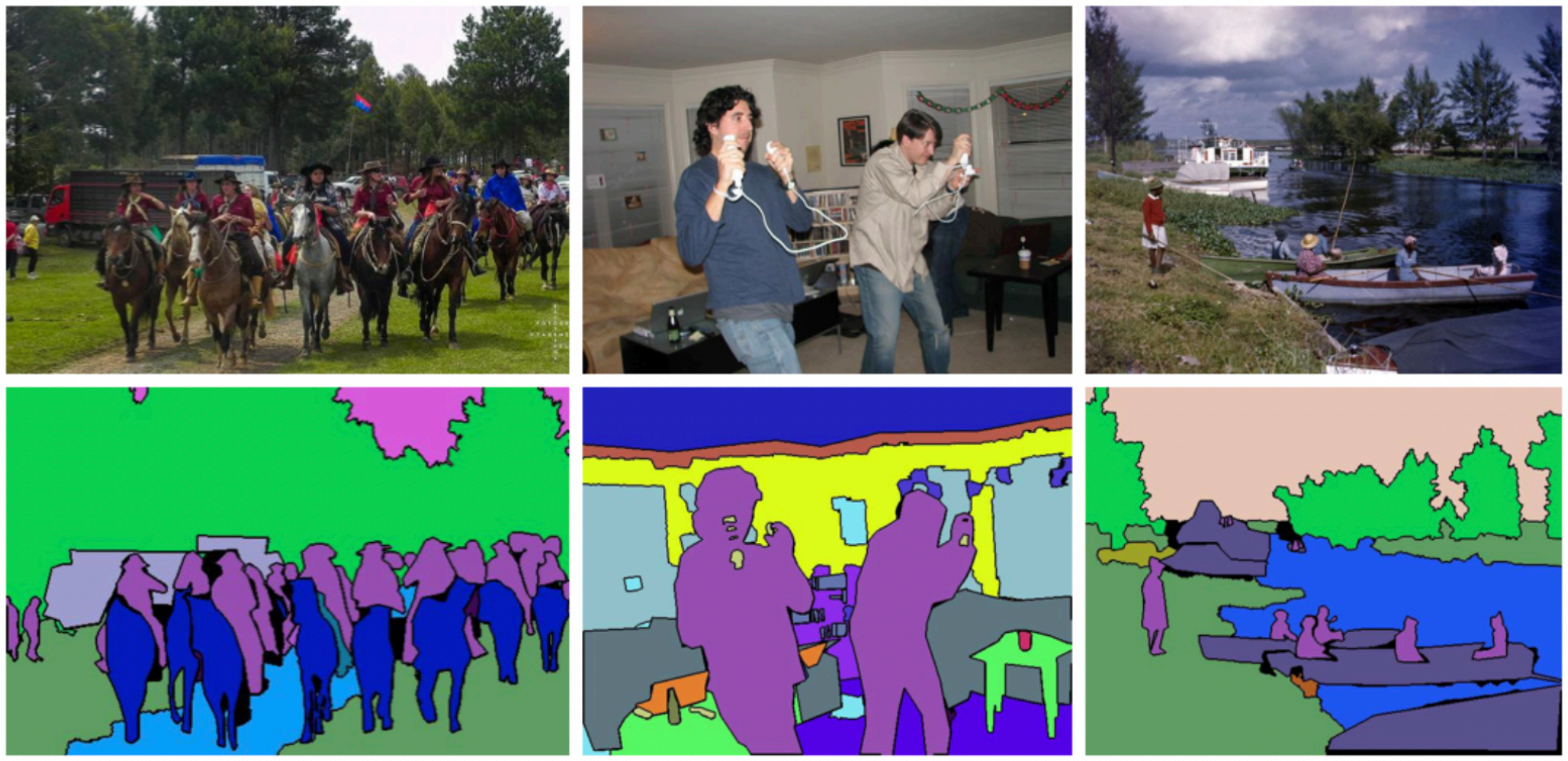
https://cocodataset.org/#home

COCO: Common Objects in Context

Tasks (competitions until 2020):

https://cocodataset.org/#home

COCO 2020 Panoptic Segmentation Task



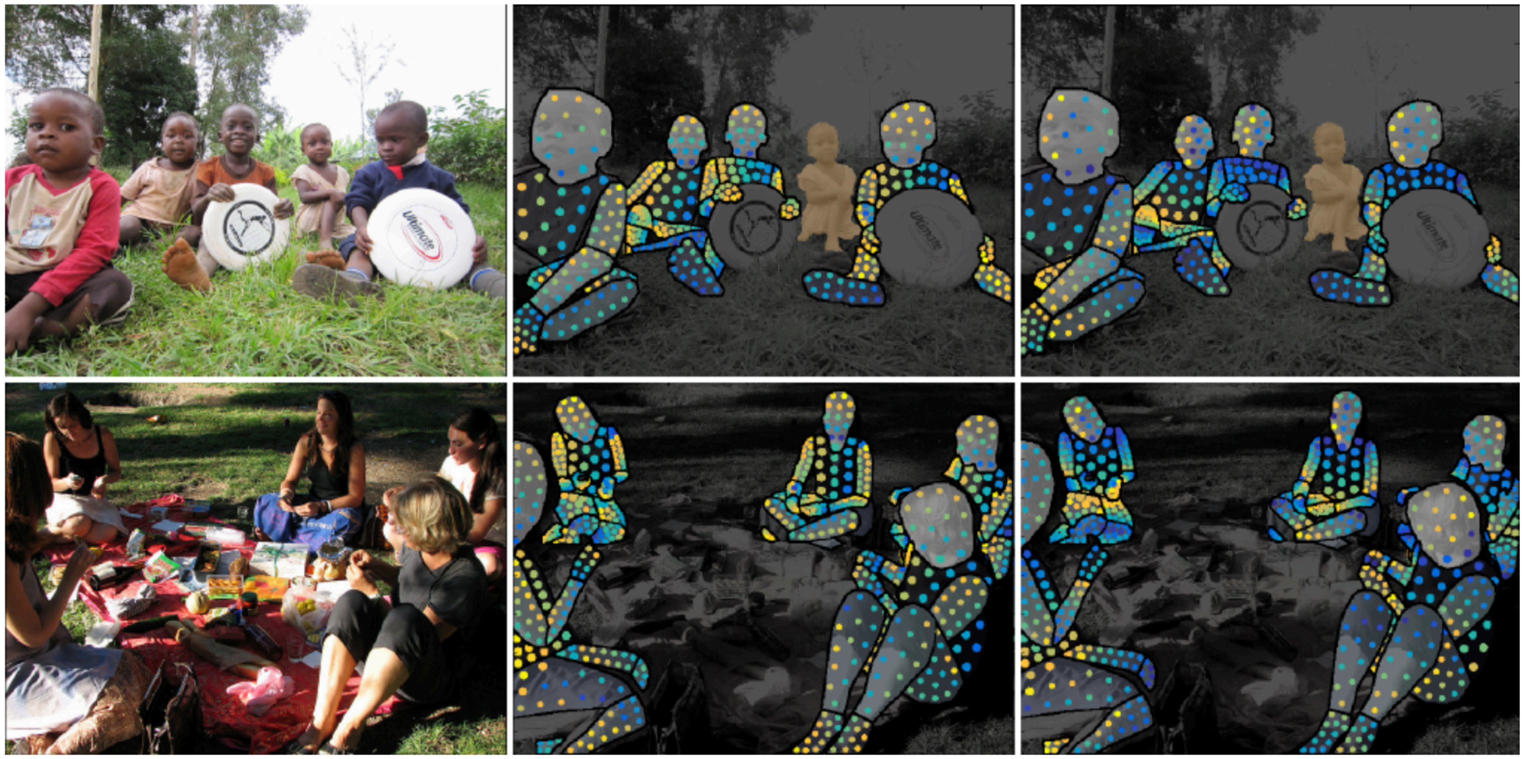
1. Overview

The COCO Panoptic Segmentation Task is designed to push the state of the art in scene segmentation. Panoptic segmentation addresses both stuff and thing classes, unifying the typically distinct semantic and instance segmentation tasks. The aim is to generate coherent scene segmentations that are rich and complete, an important step toward real-world

COCO: Common Objects in Context

Tasks (competitions until 2020):

COCO 2020 DensePose Task



1. Overview

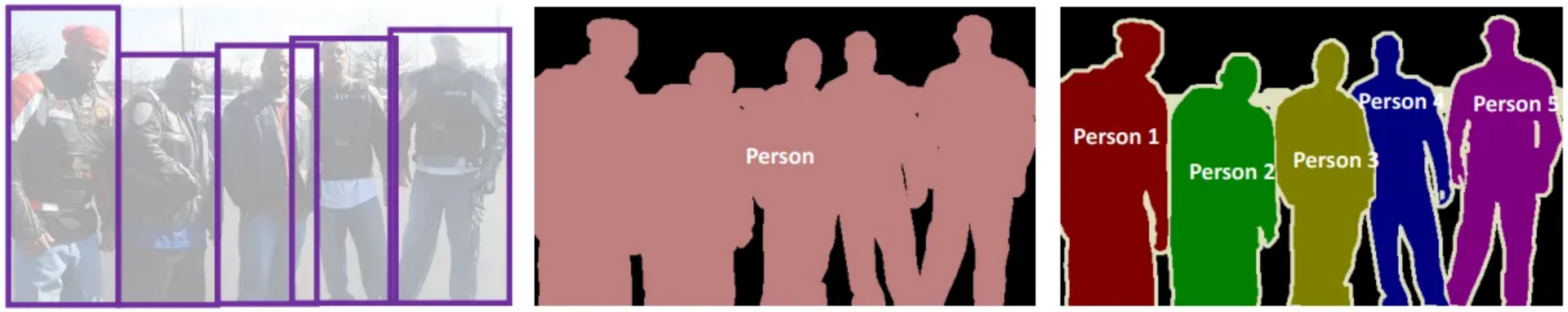
The COCO DensePose Task is designed to push the state of the art in dense estimation of human pose in challenging, uncontrolled conditions. The DensePose task involves simultaneously detecting people, segmenting their bodies and mapping all image pixels that belong to a human body to the 3D surface of the body. For full details of this task please see the DensePose evaluation page.

Detection, Localization, Segmentation

Classification is important but often times we care about where objects are in an image, or to identify multiple objects in an image.

Let’s start off with how you might do segmentation.

Segmentation



Object Detection Semantic Segmentation **Instance Segmentation**

From COCO:

“The panoptic segmentation task involves **assigning a semantic label and instance id for each pixel of an image, which requires generating dense, coherent scene segmentations.** The stuff annotations for this task come from the COCO-Stuff project described in this paper. For more details about the panoptic task, including evaluation metrics, please see the panoptic segmentation paper."

“Panotptic annotations defines defines 200 classes but only uses 133”

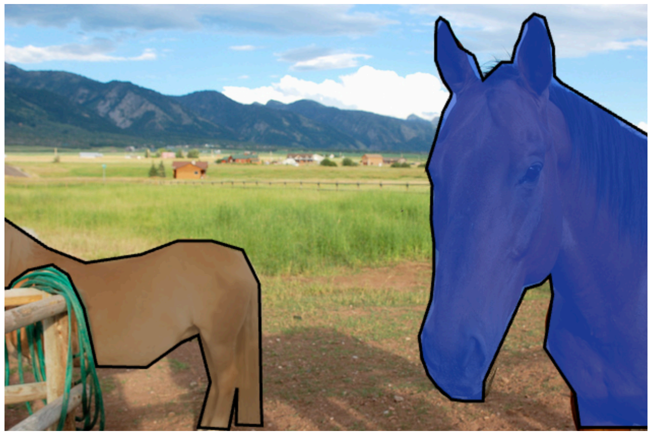
Segmentation

How do we go from

Segmentation

How do we go from

to



Segmentation

Naive idea 1: Classify each pixel?

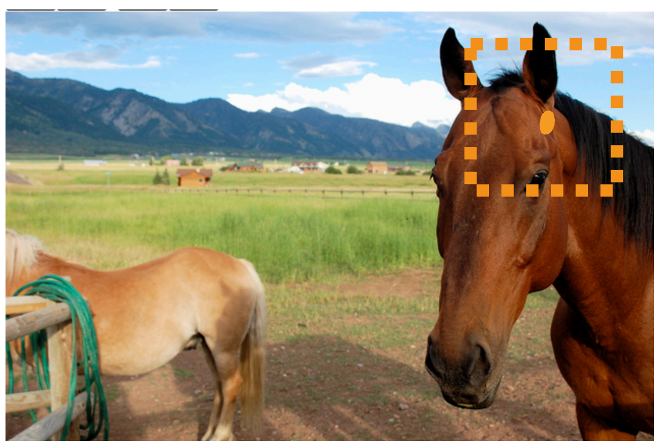
Segmentation

Naive idea 1: Classify each pixel?

Question for class: If we can’t classify each pixel, what should we do instead? What are options?

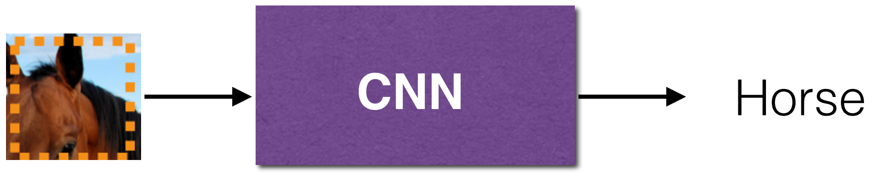
Segmentation

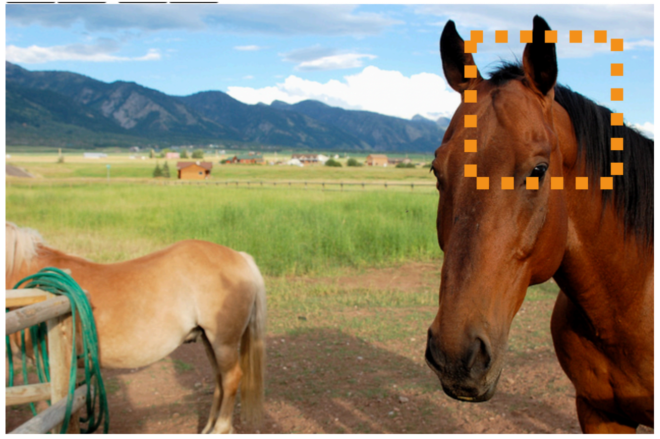
Idea 2: sliding windows?



Segmentation

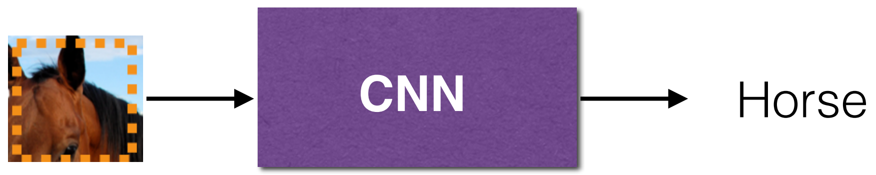
Sliding windows?

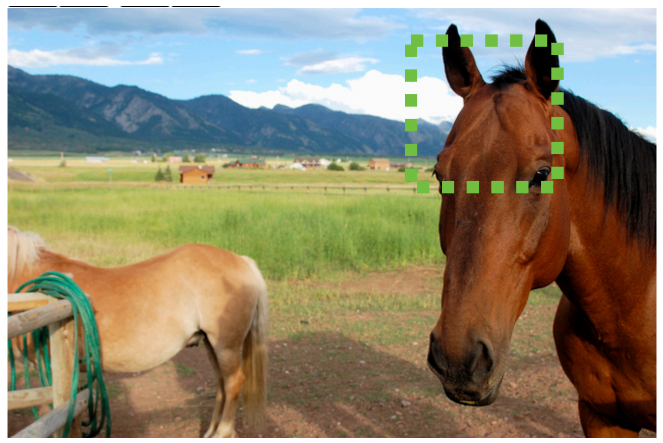




Segmentation

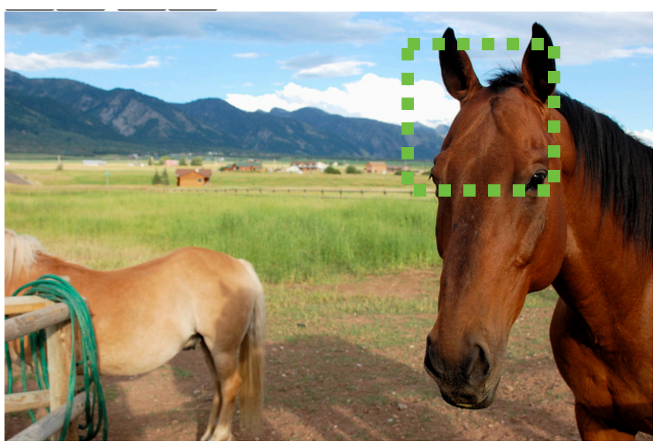
Sliding windows?

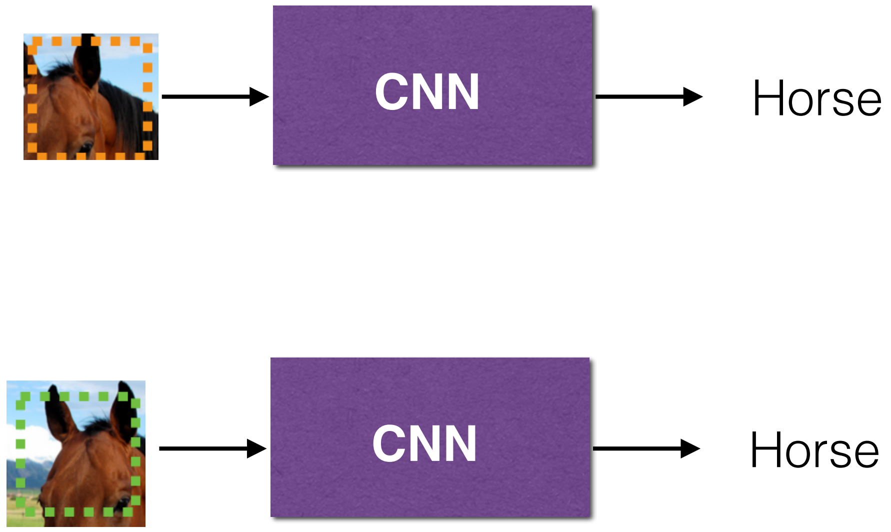




Segmentation

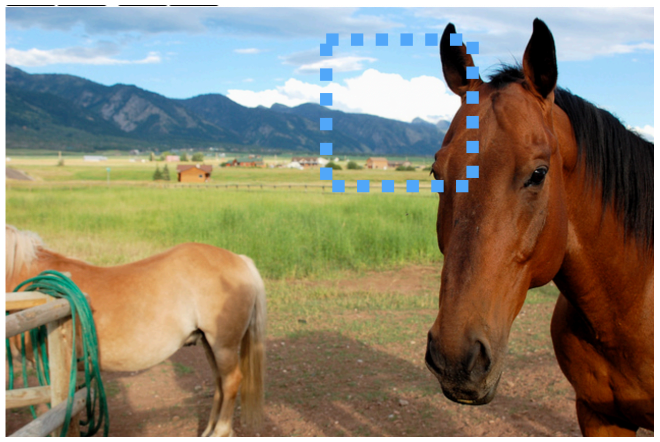
Sliding windows?

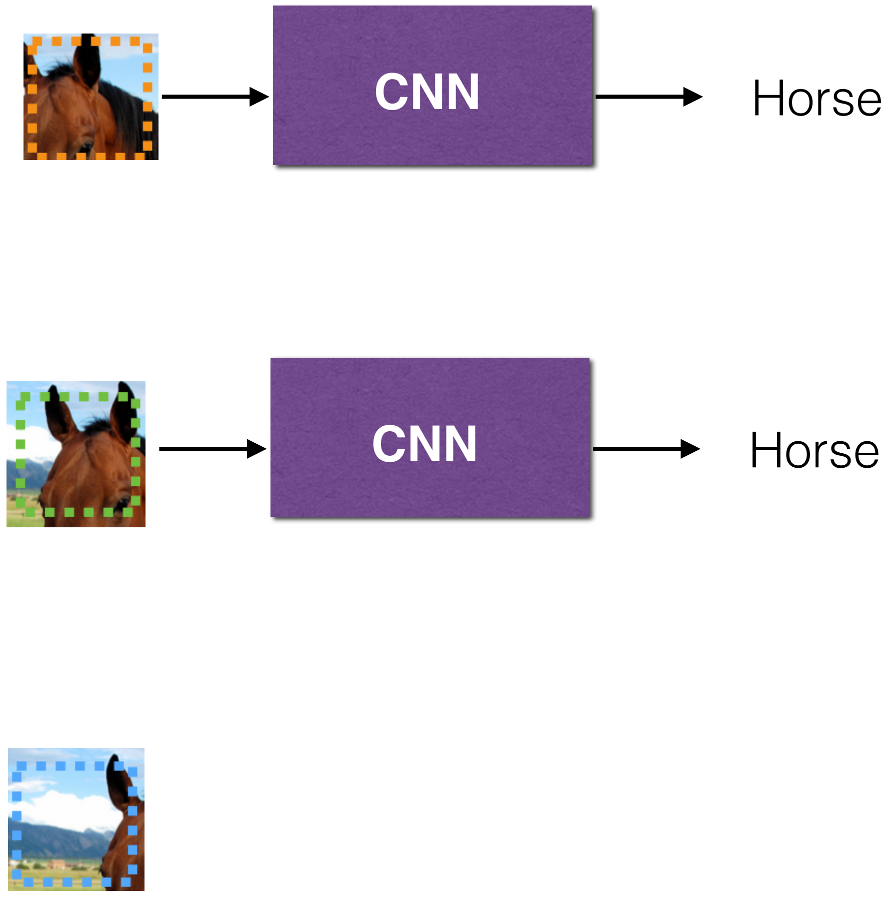




Segmentation

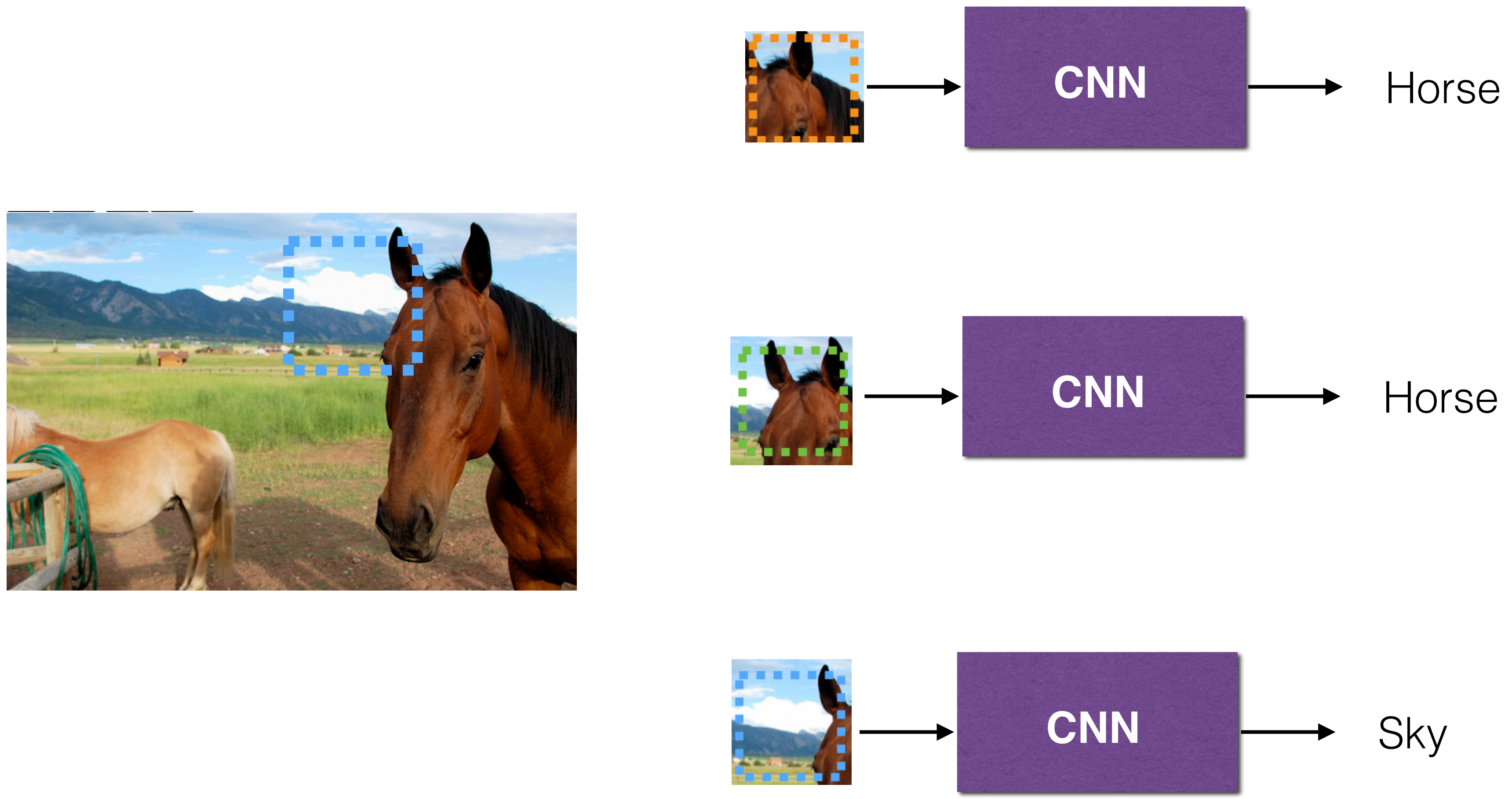
Sliding windows?





Segmentation

Sliding windows?



Question for class: What are the cons of a sliding window approach?

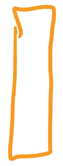
Segmentation

Cons of sliding windows?

Segmentation

Cons of sliding windows?

- Very large computational expense to slide all windows. Number of classifications (inferences, forward passes) through the neural network is the number of sliding windows.
    - Also inefficient: many of the same pixels are going into the same CNN, since the sliding windows are overlapping.
- Multiple classes in a single image.
    - Thought, why not also pass in skinny and wide windows (rectangles) for classification? (Think of input expected by CNN!)

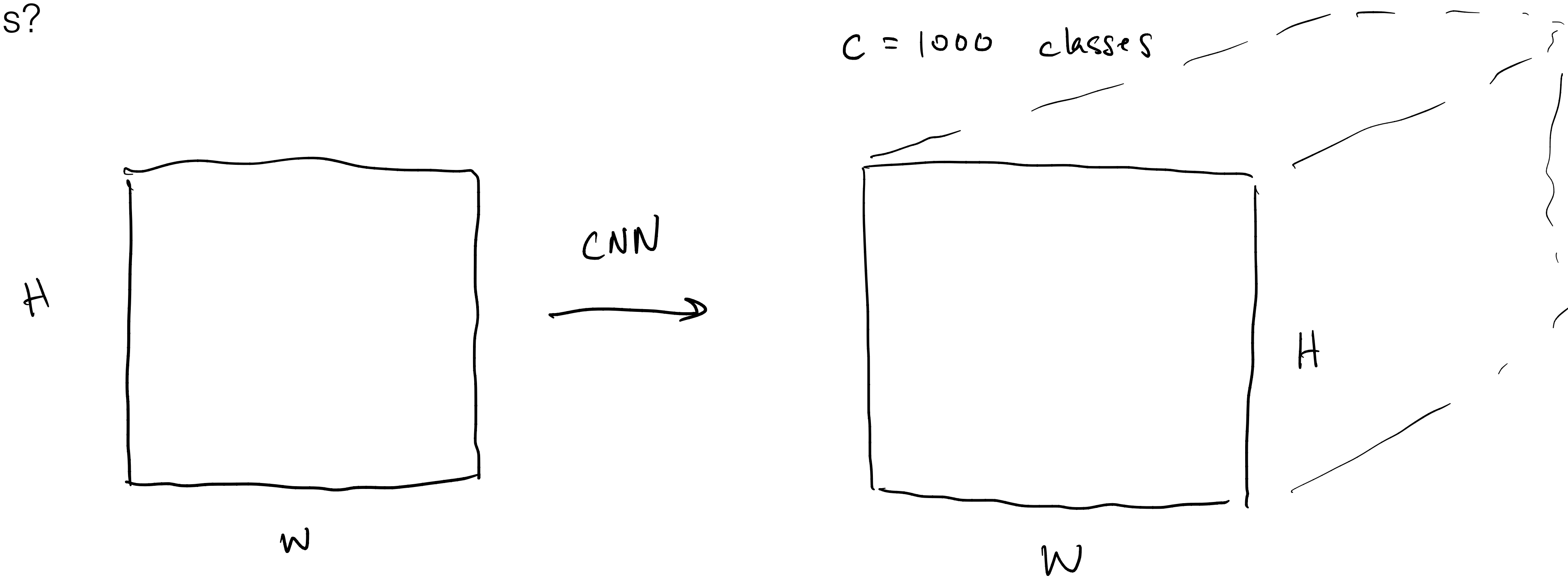


Nobody does this. You probably shouldn't, either.

Segmentation

Idea 3: why not just train one CNN to label every pixel?

Question: What are some design considerations for how you might train one CNN to do this?



Segmentation

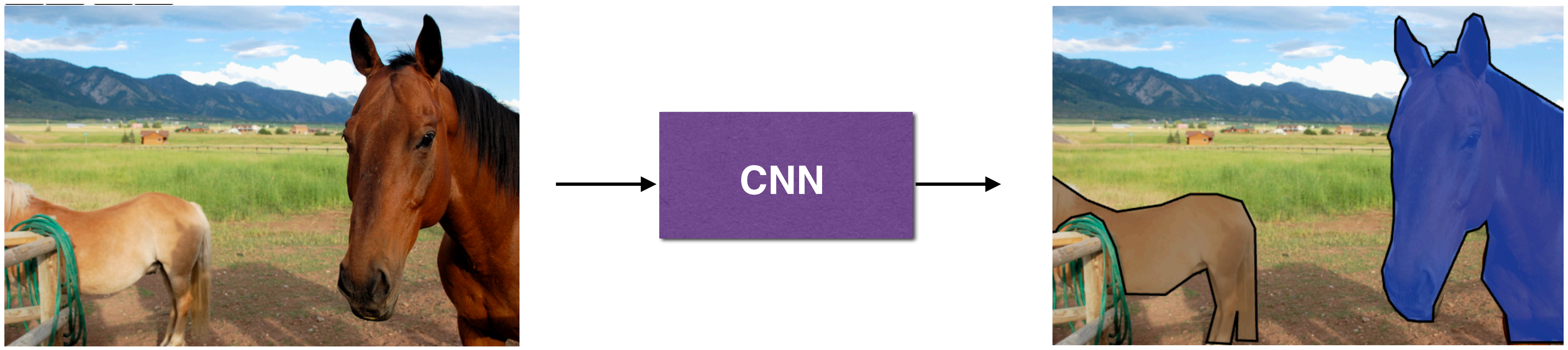
Idea 3: why not just train one CNN to label every pixel?

Question: What are some design considerations for how you might train one CNN to do this?

What kind of training data do we need for this?

Segmentation

Why not just train one CNN to do this?



What does the output size of this CNN have to be?

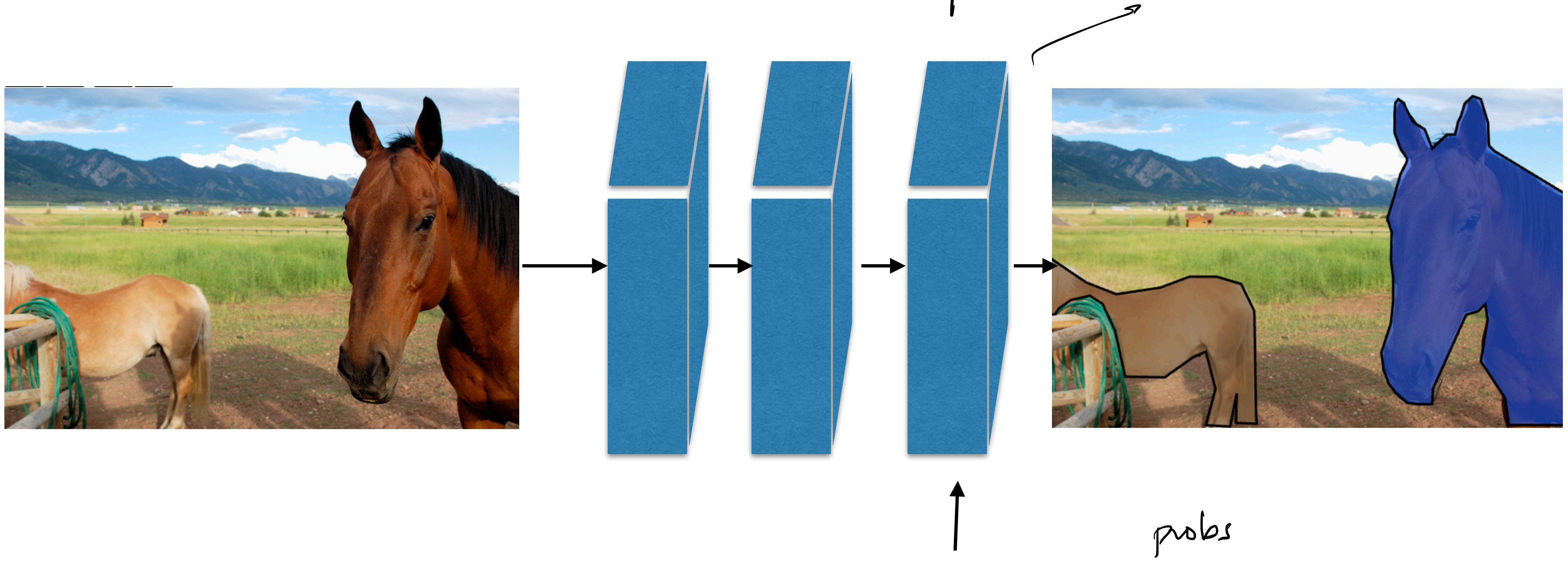
Would ResNets, GoogLeNets, VGGNets, AlexNets, work here?

Segmentation

Why not just train one CNN to do this?

$256 \cdot 256 \cdot 1000 = 65{,}536{,}000$ #'s

4 bytes $\Rightarrow$ 250MB

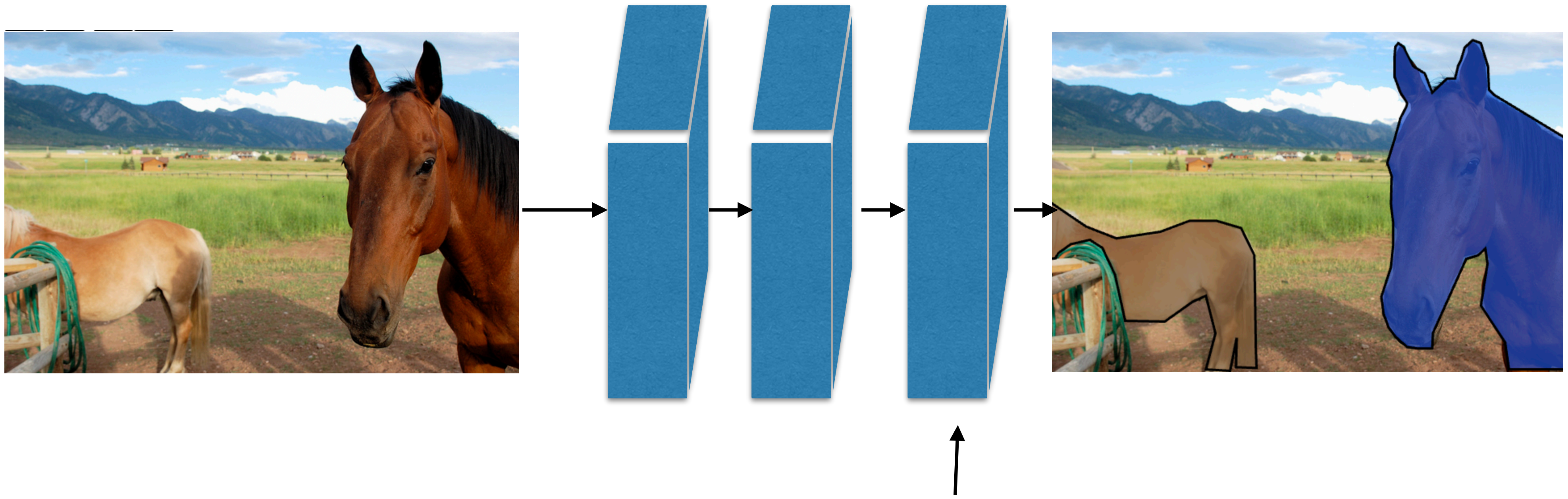


This should be a C x H x W tensor of ~~scores~~ probs per pixel.

Question: What is the major con of this approach?

Segmentation

Why not just train one CNN to do this?

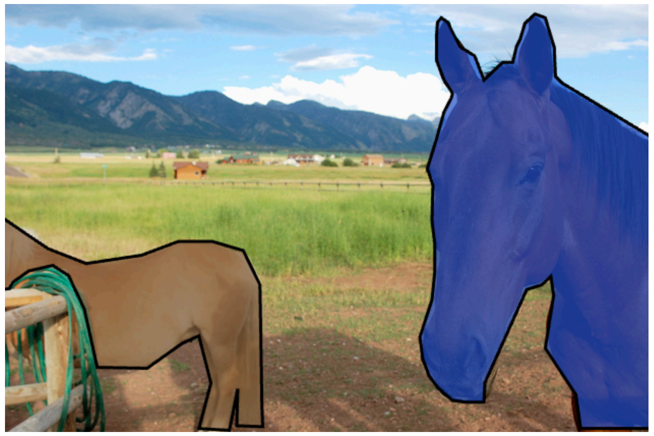


This should be a C x H x W tensor of scores per pixel.

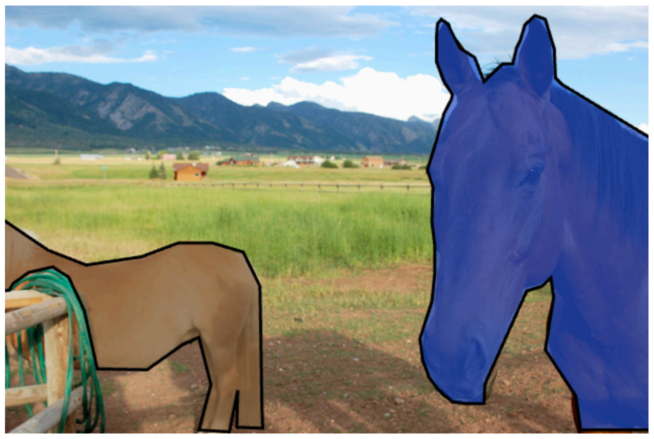
Cons?

Images are large, there will be significant memory + computational expense.

Have a bottleneck to reduce the number of parameters

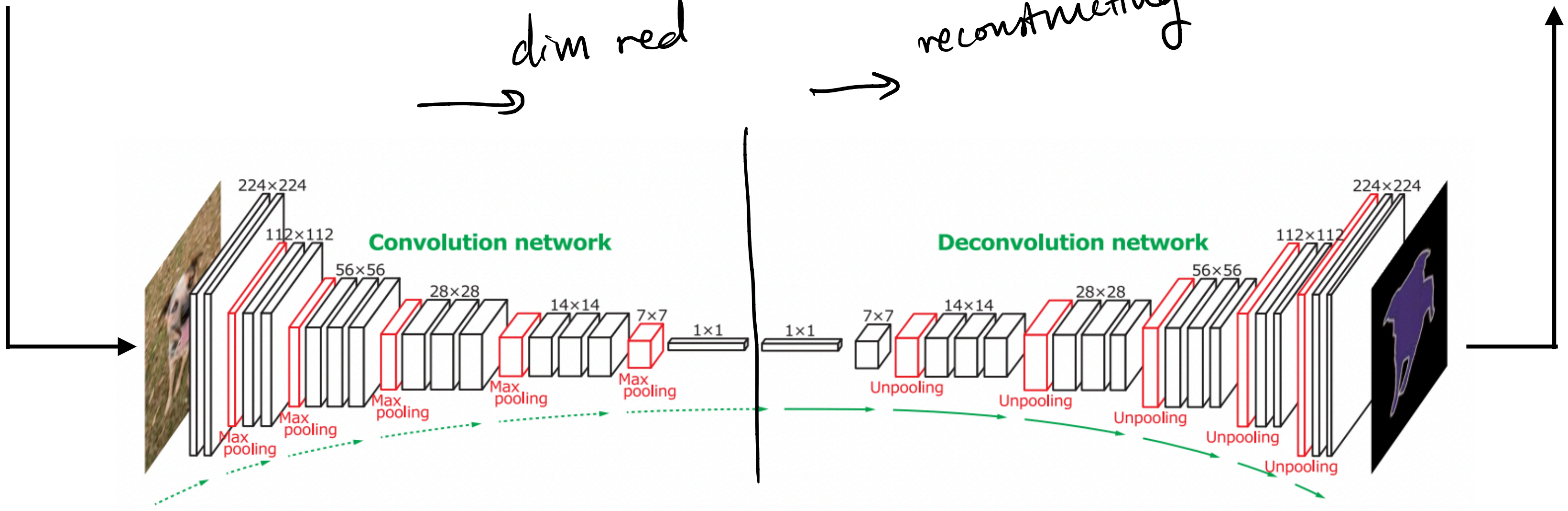


Have a bottleneck to reduce the number of parameters



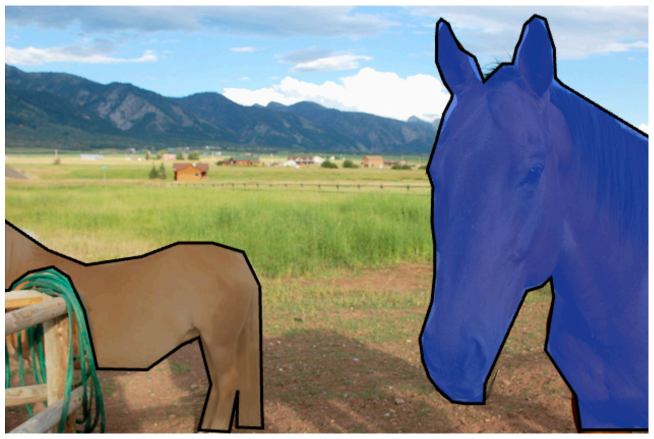
dim red $\longrightarrow$

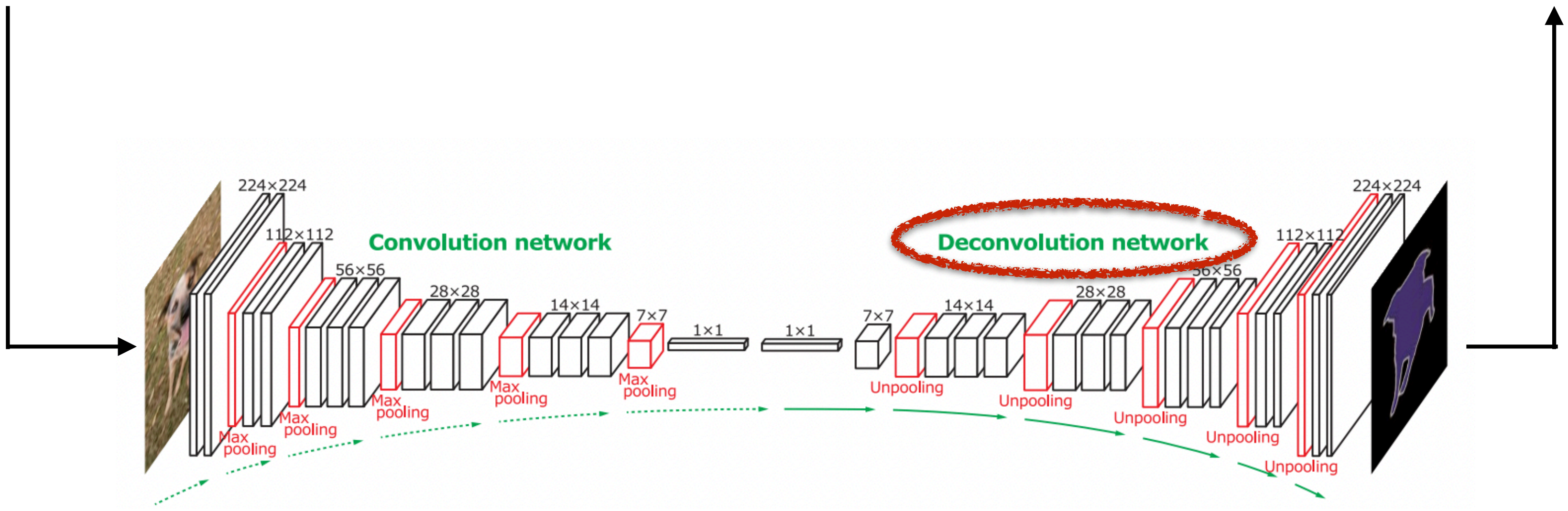
$\longrightarrow$ reconstructing



https://arxiv.org/pdf/1505.04366.pdf

Have a bottleneck to reduce the number of parameters

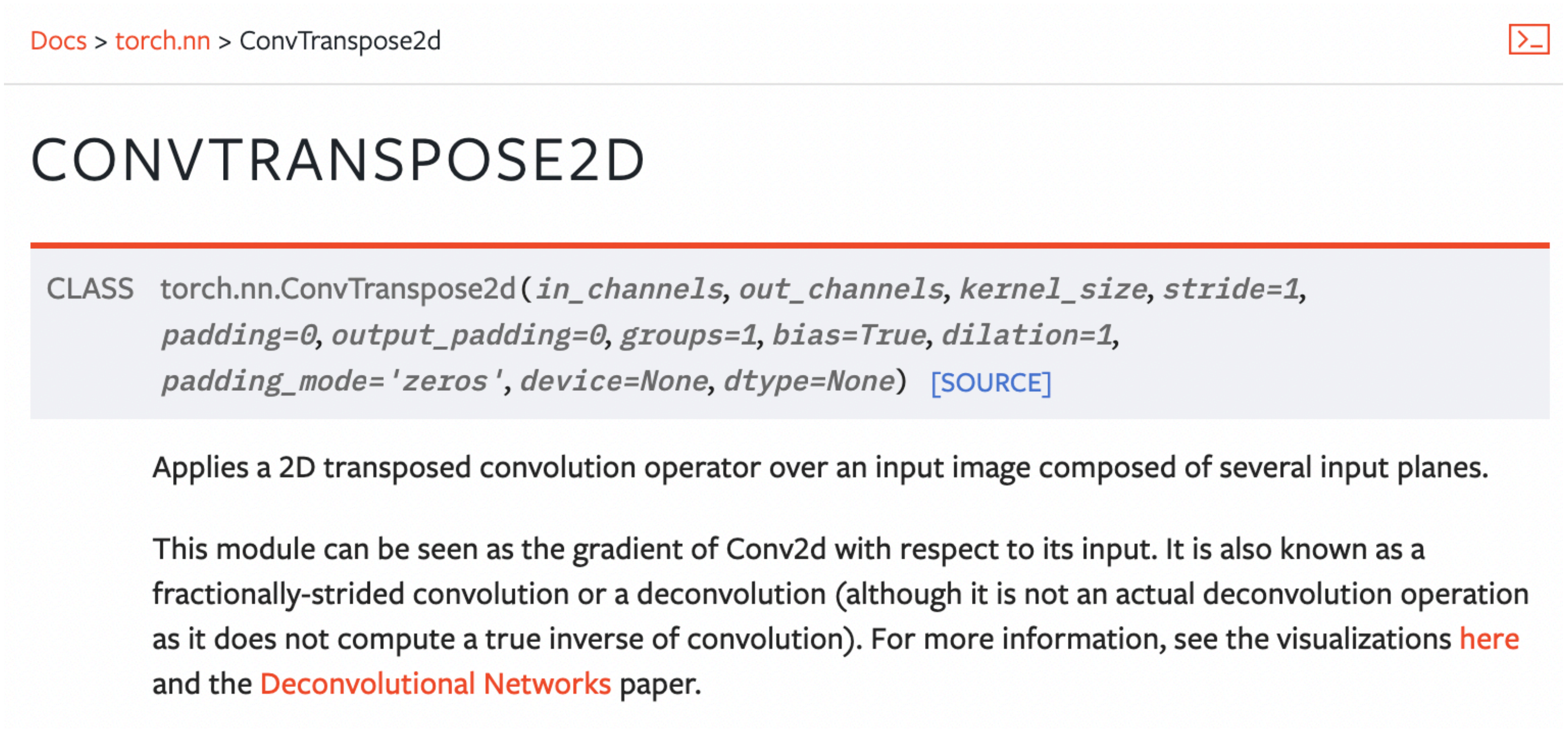




https://arxiv.org/pdf/1505.04366.pdf

Aside: DO NOT CALL THIS DECONVOLUTION. This is called TRANSPOSED CONV.

Transposed convolution


Docs > torch.nn > ConvTranspose2d

CONVTRANSPOSE2D

CLASS torch.nn.ConvTranspose2d(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*, *device=None*, *dtype=None*) [SOURCE]

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation as it does not compute a true inverse of convolution). For more information, see the visualizations here and the Deconvolutional Networks paper.

Unpool?

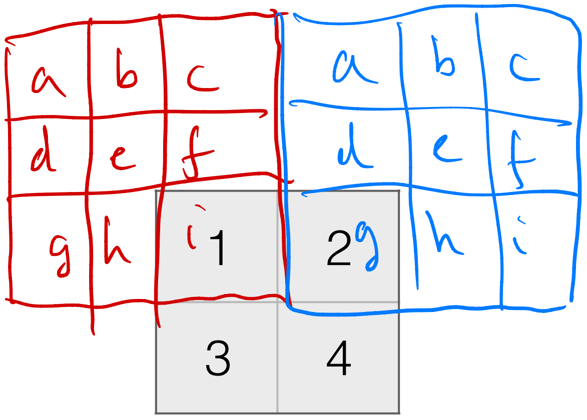
2x2 max pool
stride 2

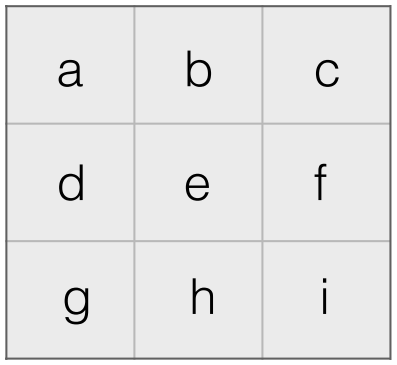
1	2
3	4

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

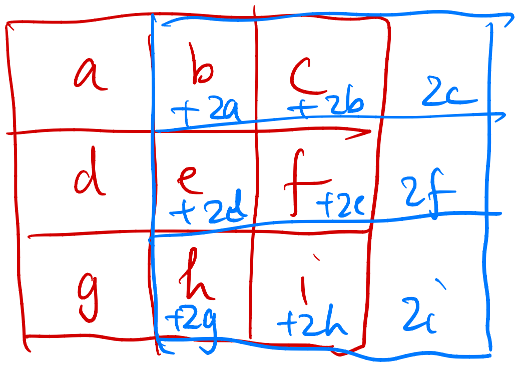
Transposed convolution

stride = 1



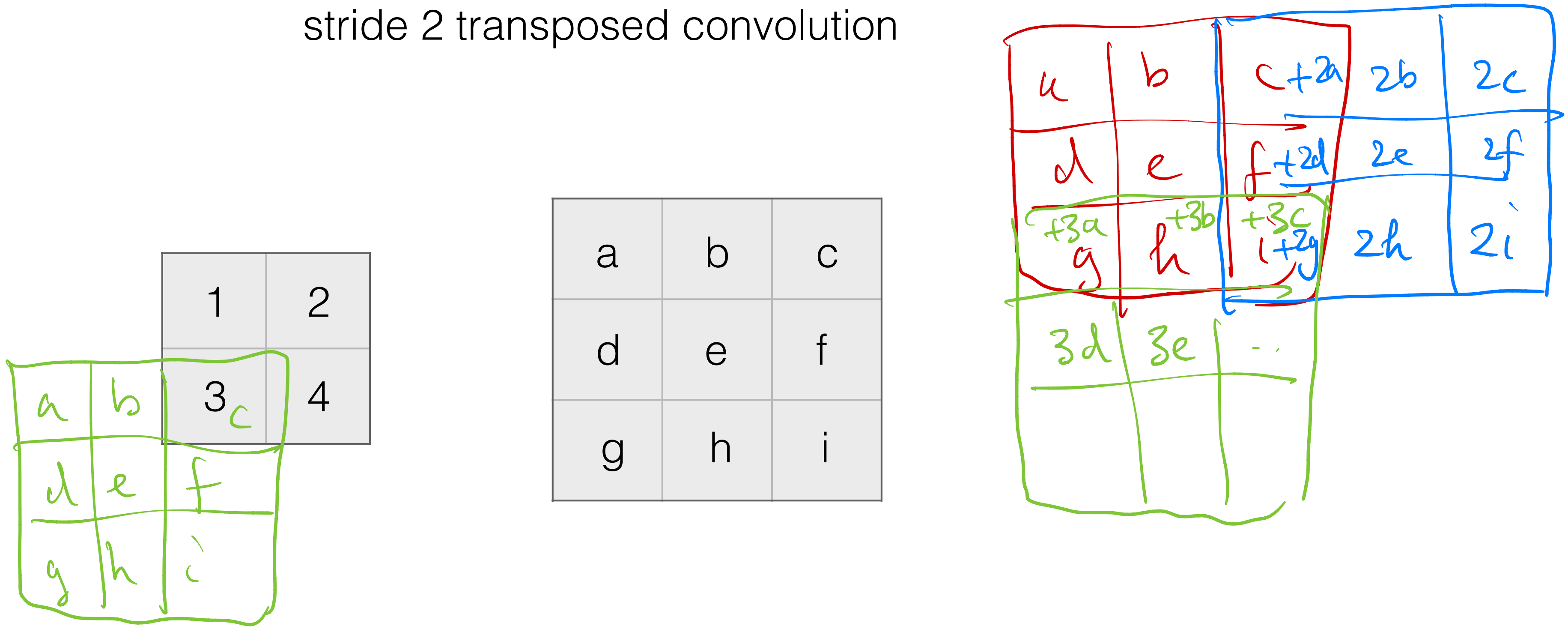


a	b	c
d	e	f
g	h	i



Transposed convolution

stride 2 transposed convolution



Transposed convolution

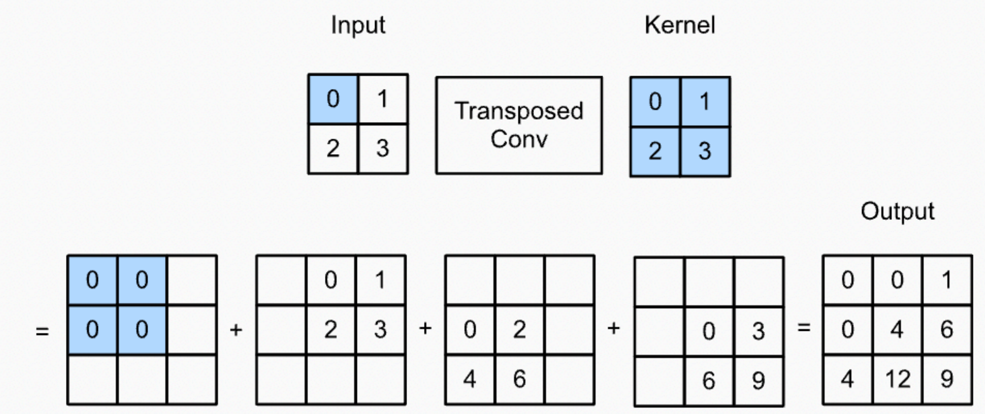


Fig. 14.10.1 Transposed convolution with a 2×2 kernel. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation.

https://d2l.ai/chapter_computer-vision/transposed-conv.html

Transposed convolution

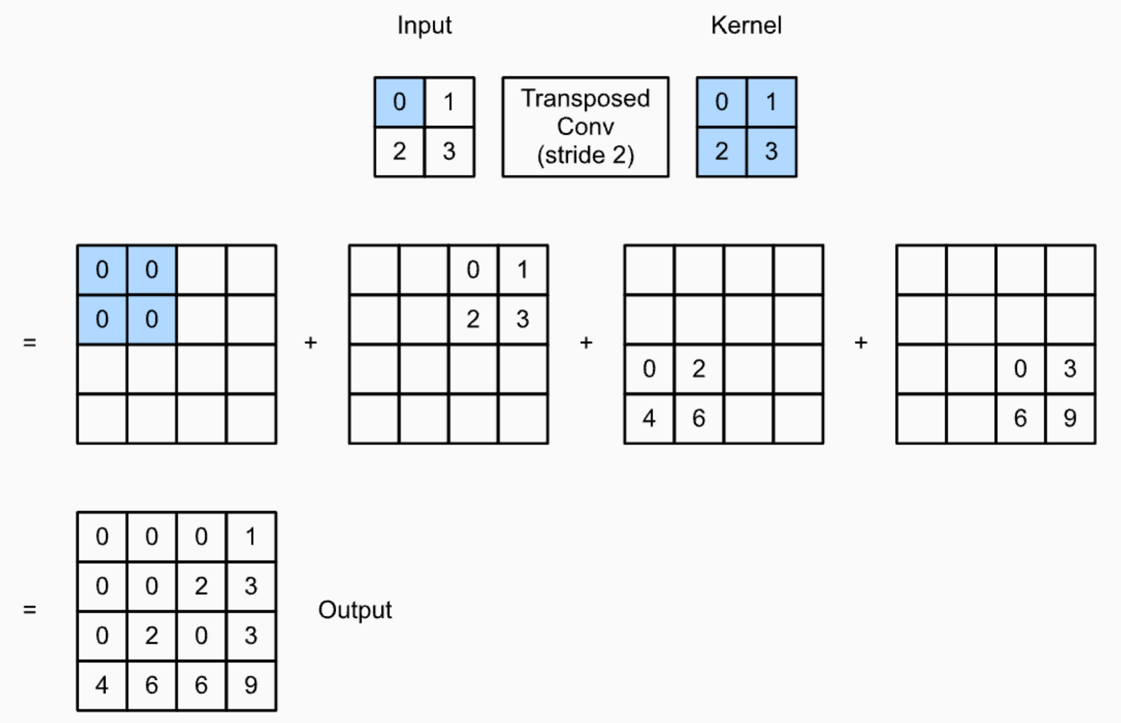
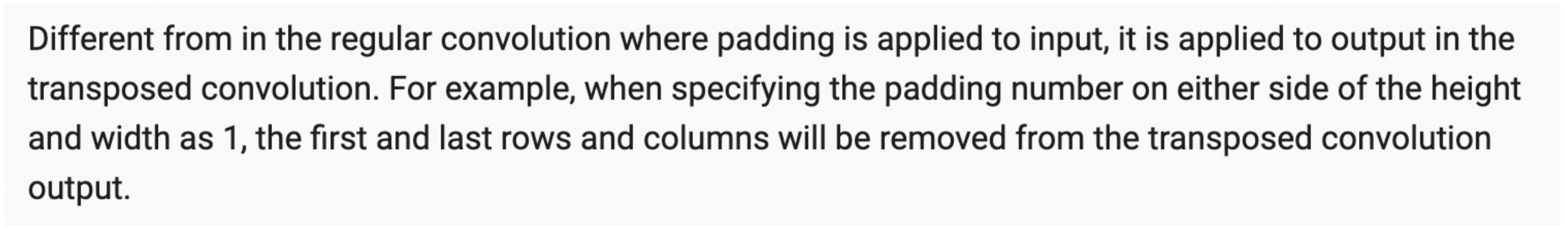


Fig. 14.10.2 Transposed convolution with a 2×2 kernel with stride of 2. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation.

https://d2l.ai/chapter_computer-vision/transposed-conv.html

Transposed convolution — padding removes from the output


Different from in the regular convolution where padding is applied to input, it is applied to output in the transposed convolution. For example, when specifying the padding number on either side of the height and width as 1, the first and last rows and columns will be removed from the transposed convolution output.

The output of a transposed convolution is therefore:

(input_size - 1) * stride + kernel_size - 2*pad

https://d2l.ai/chapter_computer-vision/transposed-conv.html

One more padding example

What is the size of an output of a transposed convolution with:

Input: 2x2
Kernel: 3x3
Stride = 2
Pad = 1

From equation, (input_size - 1) * stride + kernel_size - 2*pad = 2 + 3 - 2 = 3.