Zurich University of Applied Sciences

NN for sequential data cntd.

&

Quantify prediction uncertainties

Beate Sick

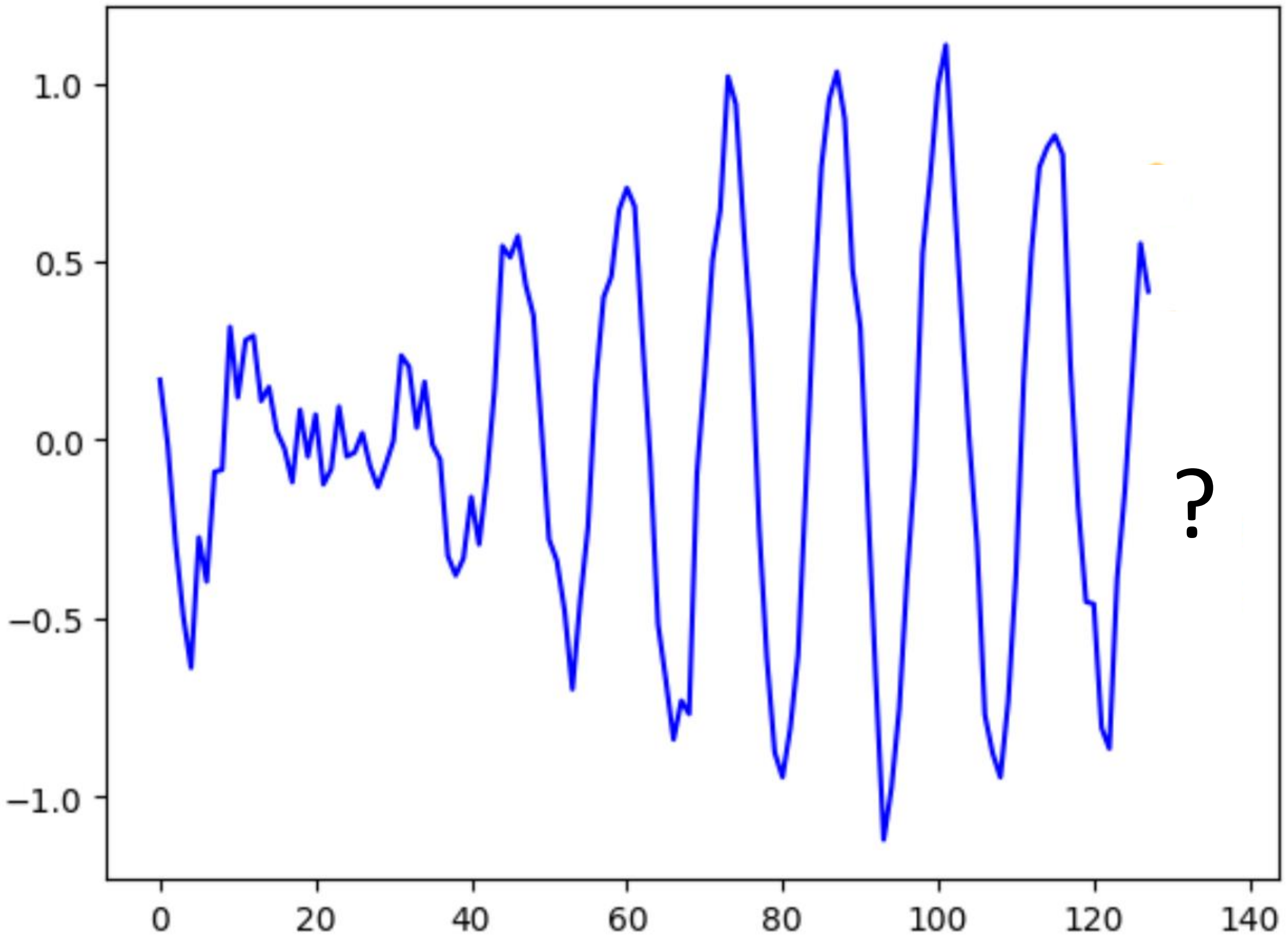
sick@zhaw.ch

Remark: Much of the material has been developed together with Elvis Murina and Oliver Dürr

Topics

- **Recap architectures for sequential data**
  - 1D convolution
  - RNN
- **Recurrent NN with better memory**
  - GRU
  - LSTM
- **How reliable are predicted probablilities?**
  - Calibration
  - Quantifying prediction uncertain via Dropout

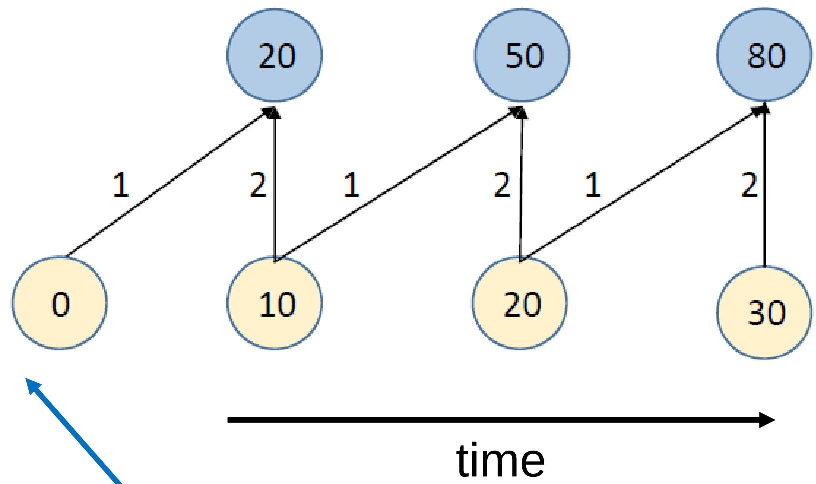
Task in homework: Predict how series will continue



Recap 1D “causal” convolution for ordered data

Toy example:

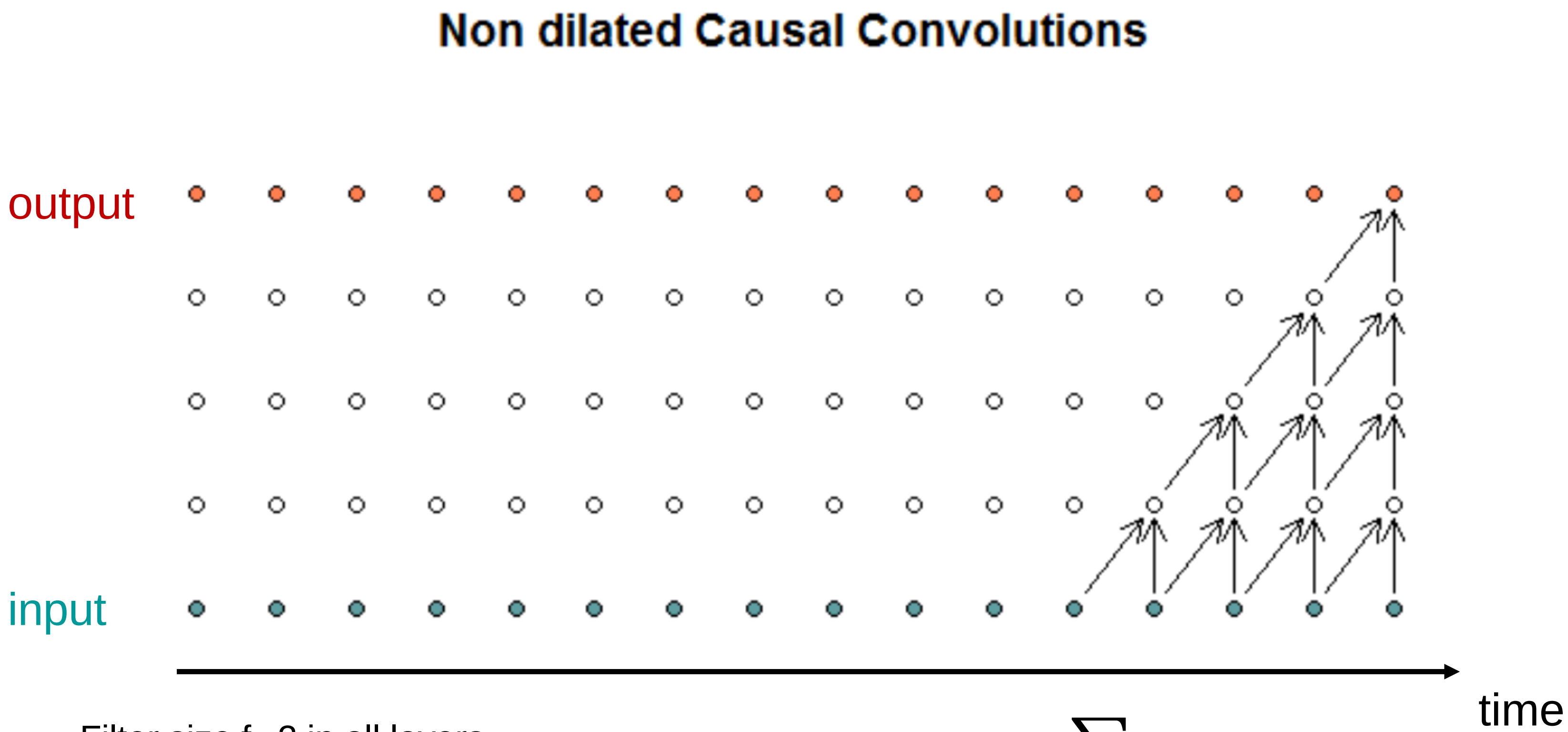
Output:	23	50	80
1D Kernel:	1	2	
Input	10	20	30



To make all layers the same size, a zero padding is added to the beginning of the input layers

“causal” networks, because the architecture ensured that no information from the future is used.

Stacking 1D "causal" convolutions without dilation



Filter size f_i=2 in all layers.
Hidden layers from i=1…3

$$ReceptiveField = f_0 + \sum_i (f_i - 1)$$

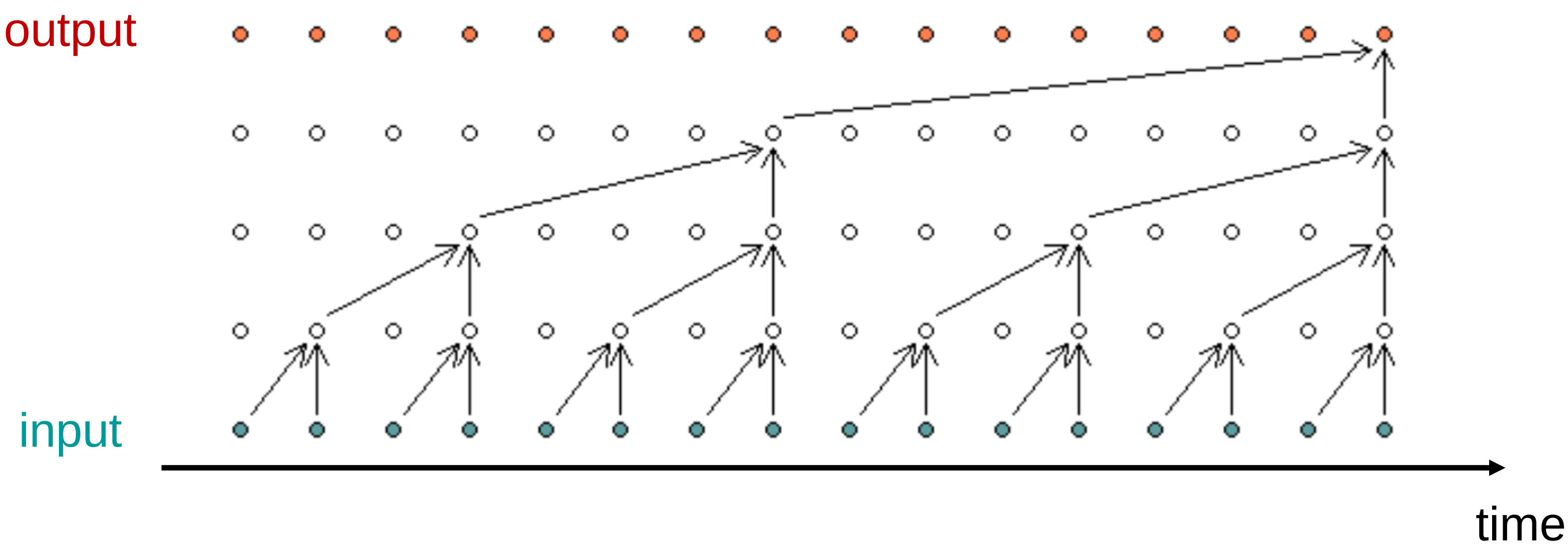
Stacking k causal 1D convolutions with kernel size 2 allows to look back k time-steps.

After 4 layers each neuron has a "memory" of 5 time-steps (1 presence and 4 past).

https://deepmind.com/blog/wavenet-generative-model-raw-audio/

Dilation allows to increase "memory" = receptive field

To increase the memory of neurons in the output layer, you can use "dilated" convolutions:



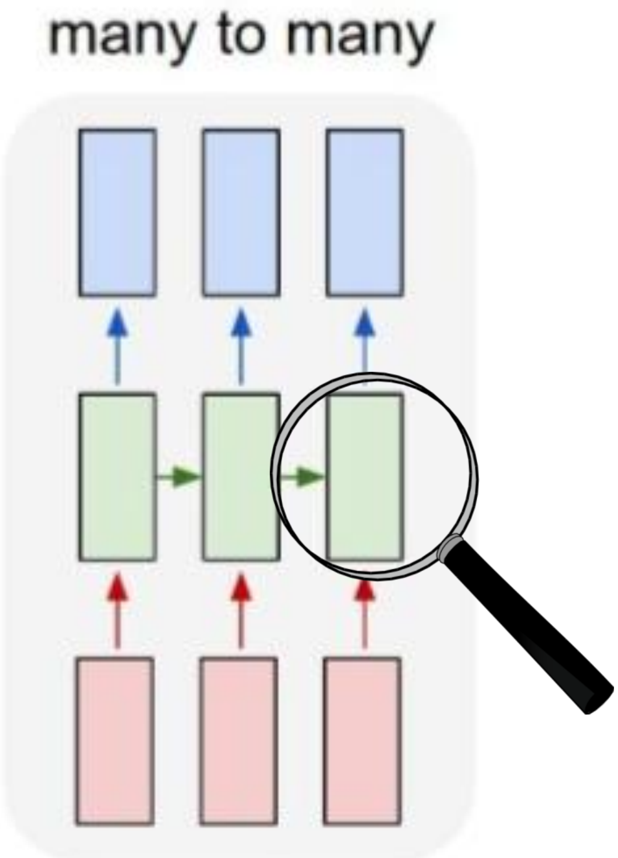
$$ReceptiveField = f_0 \cdot d_0 + \sum_i (f_i - 1) \cdot d_i \; = 2 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 = 16$$

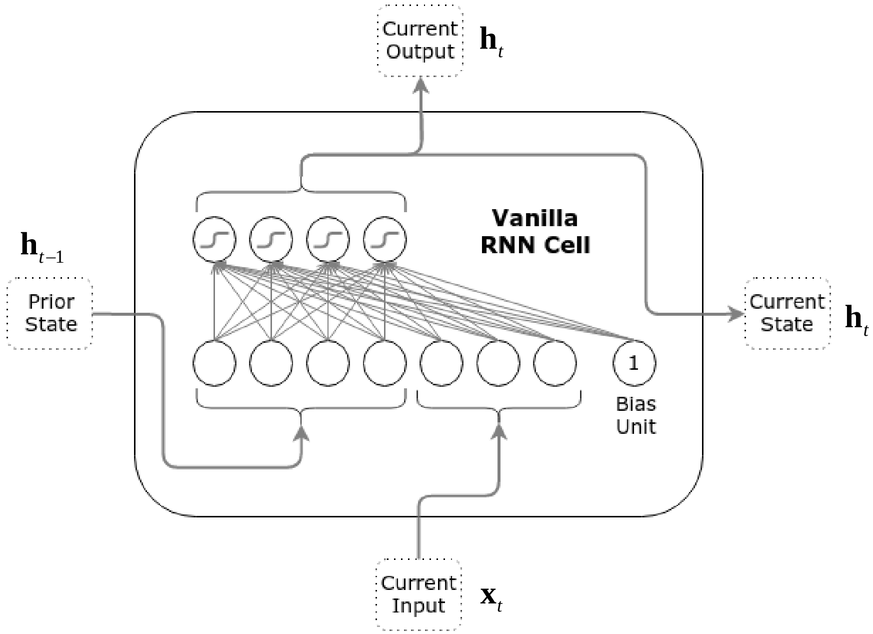
Here the filter f_i=2 for all layer, but dilation is d_i starts with 1 and doubles from layer to layer

After 4 layers each neuron has a receptive field of 16 input neurons.

https://deepmind.com/blog/wavenet-generative-model-raw-audio/
https://fomoro.com/projects/project/receptive-field-calculator Receptive field calculator

Recap the architecture of a simple RNN



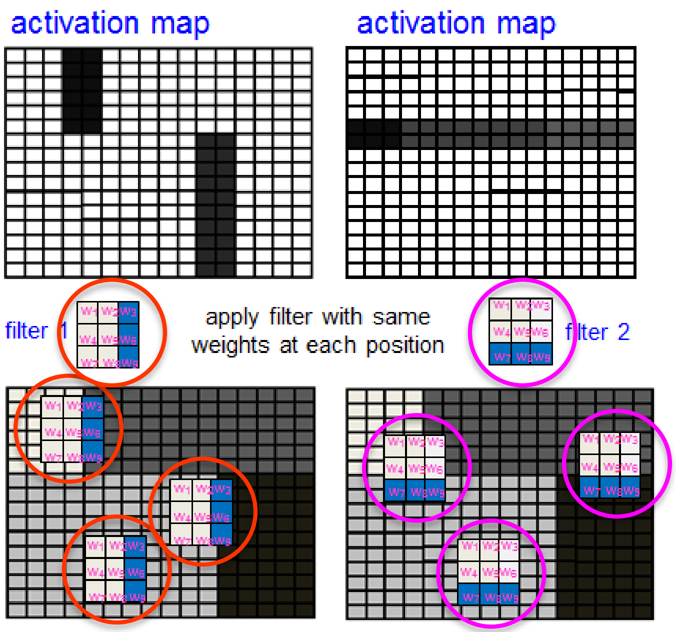


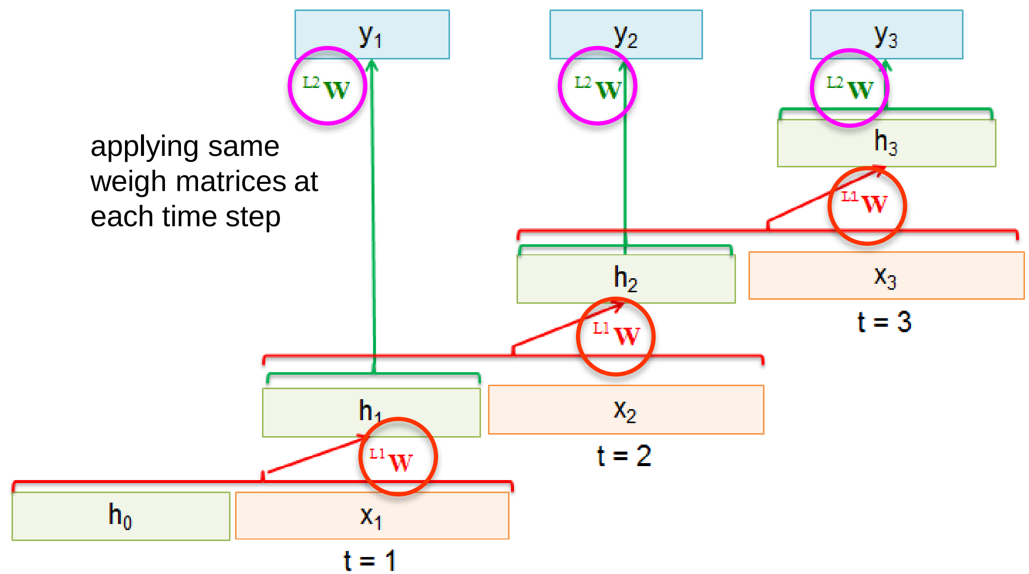
$$\text{output}=\mathbf{h}_t = \tanh\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W} + \mathbf{b}\right)$$

image credits: https://r2rt.com/written-memories-understanding-deriving-and-extending-the-lstm.html

Common tricks in RNN & CNN and some differences

CNN and Recurrent Network share weights



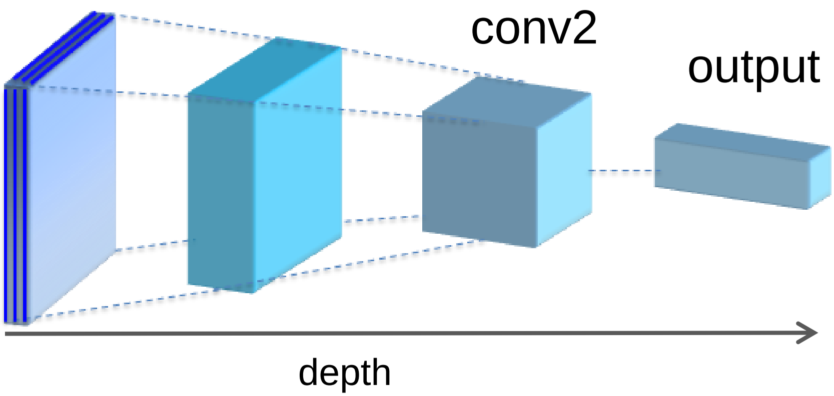


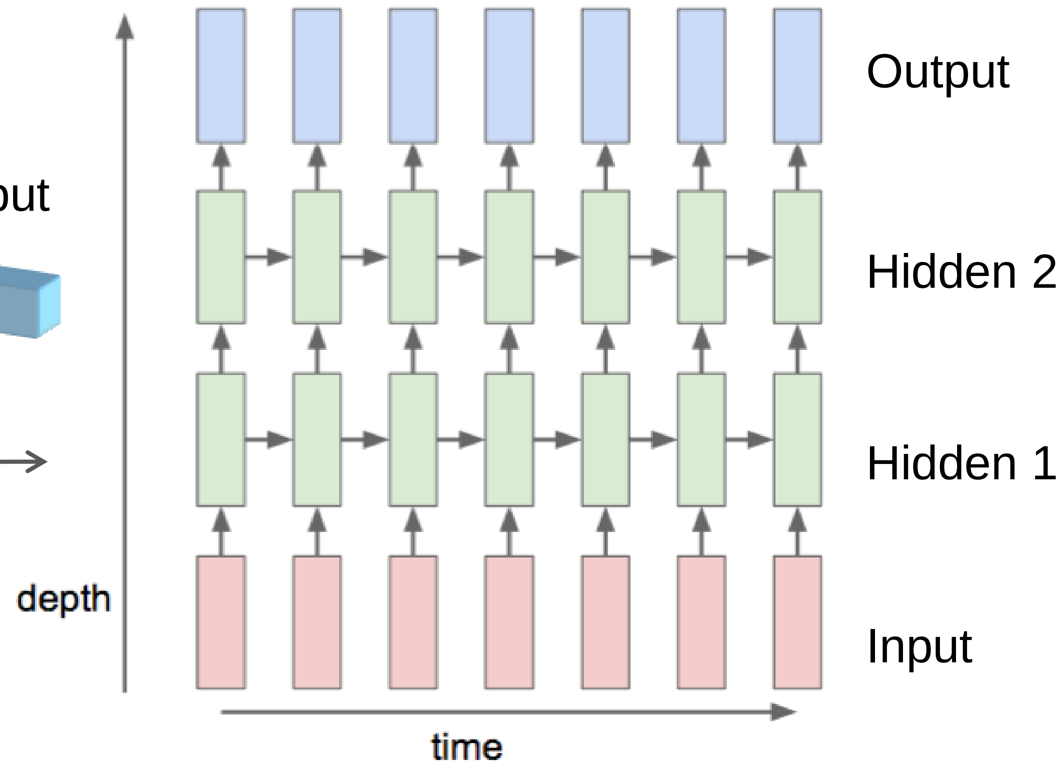
CNN share weights between different local regions of the image

RNN share weights between time steps

Remark: no weight sharing in fully connected NN

Also in RNN we can go deep for hierarchical features

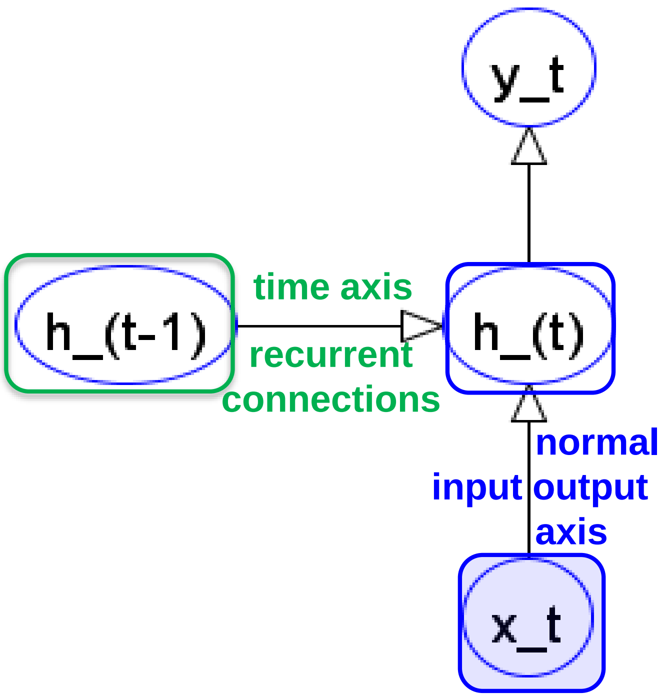


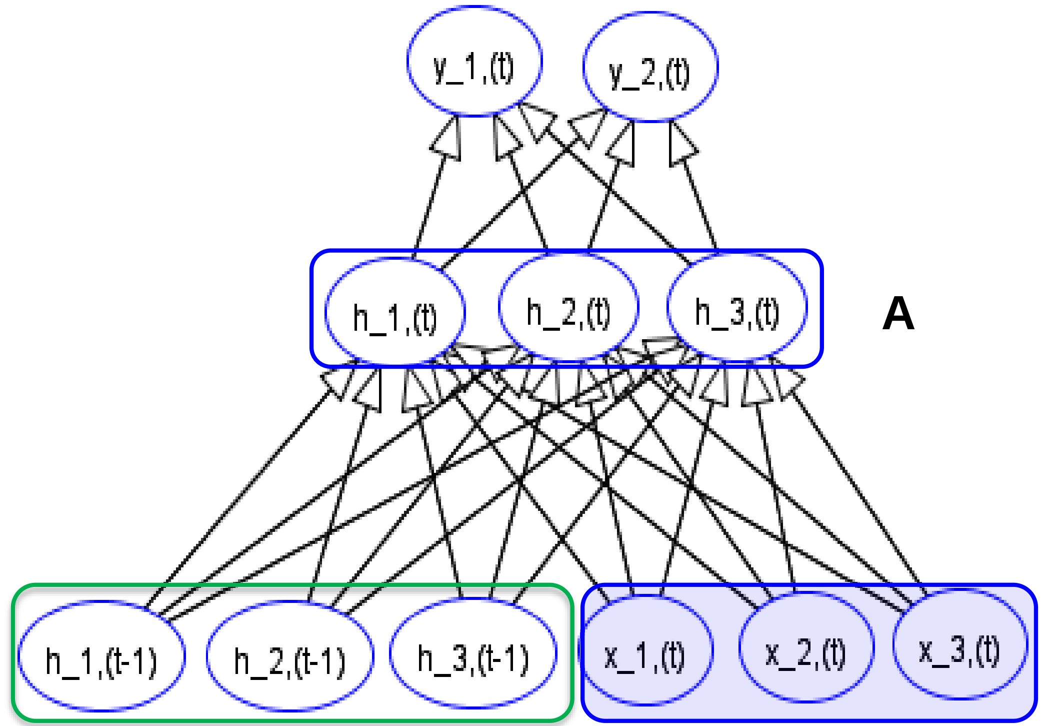


Usually we see only 1-4 hidden layers in an RNN compared to usually 4-100 stacked hidden convolutional blocks in CNNs.

Illustration: http://www.deeplearningbook.org/ and : http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Dropout in recurrent architectures allow to choose different different dropout rates for recurrent and normal connections





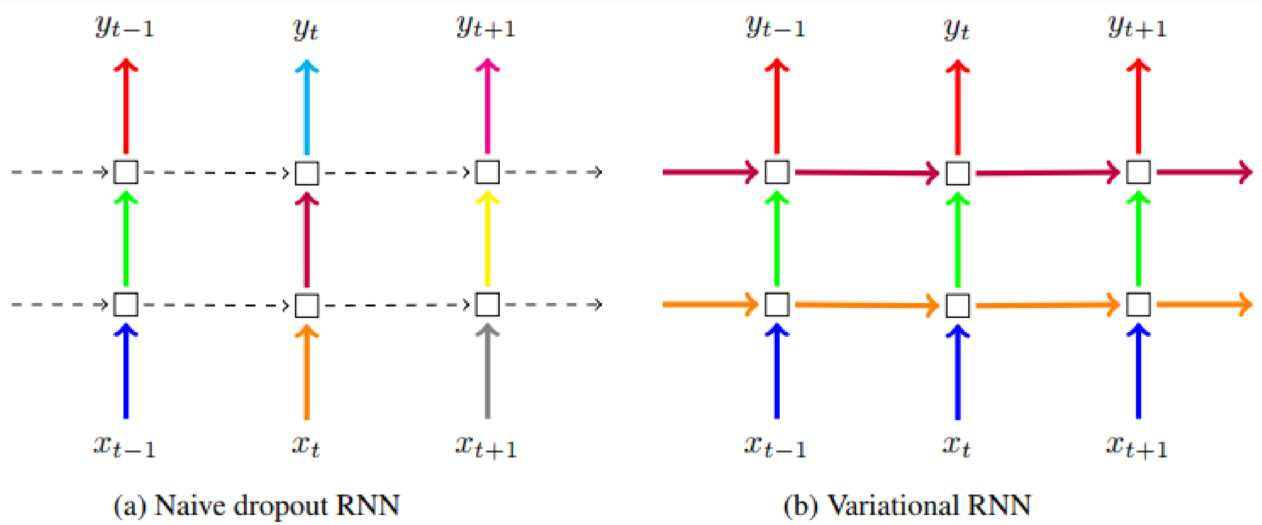
$$\mathbf{A} = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t) = \tanh\left([\mathbf{h}_{t-1}, \mathbf{x}_t]\cdot \mathbf{W} + \mathbf{b}\right) = \tanh\left(\mathbf{h}_{t-1}\cdot \mathbf{W}_h + \mathbf{x}_t \cdot \mathbf{W}_x + \mathbf{b}\right)$$

$$\mathbf{W} = \begin{pmatrix} \mathbf{W}_h \\ \mathbf{W}_x \end{pmatrix}$$

Dimensions in example: **W:**6x3, $\mathbf{W}_h$:3x3, $\mathbf{W}_x$:3x3

Dropout in recurrent architectures

It is important to use identical dropout masks (marked by arrows with same color) at different time steps in recurrent architectures like GRU or LSTM.



same arrow color indicates identical dropout mask

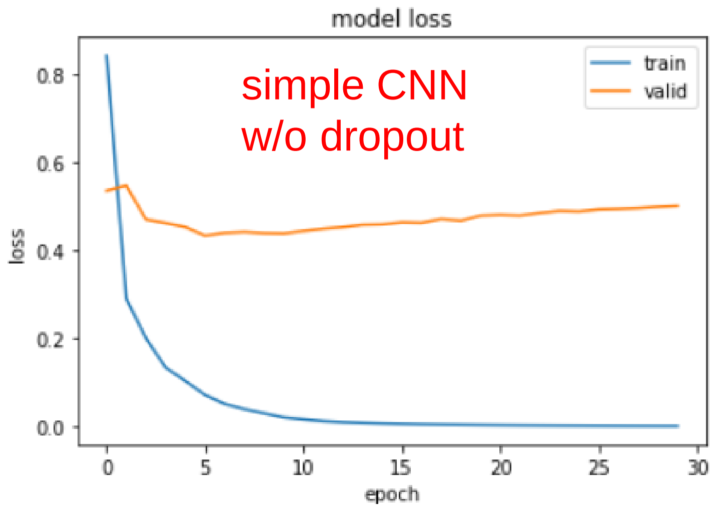
Figure 1: **Depiction of the dropout technique following our Bayesian interpretation (right) compared to the standard technique in the field (left).** Each square represents an RNN unit, with horizontal arrows representing time dependence (recurrent connections). Vertical arrows represent the input and output to each RNN unit. Coloured connections represent dropped-out inputs, with different colours corresponding to different dropout masks. Dashed lines correspond to standard connections with no dropout. Current techniques (naive dropout, left) use different masks at different time steps, with no dropout on the recurrent layers. The proposed technique (Variational RNN, right) uses the same dropout mask at each time step, including the recurrent layers.

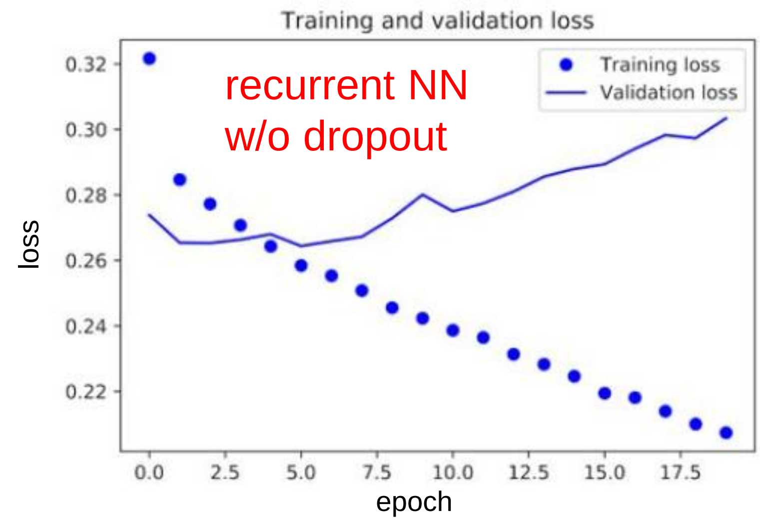
Gal2016

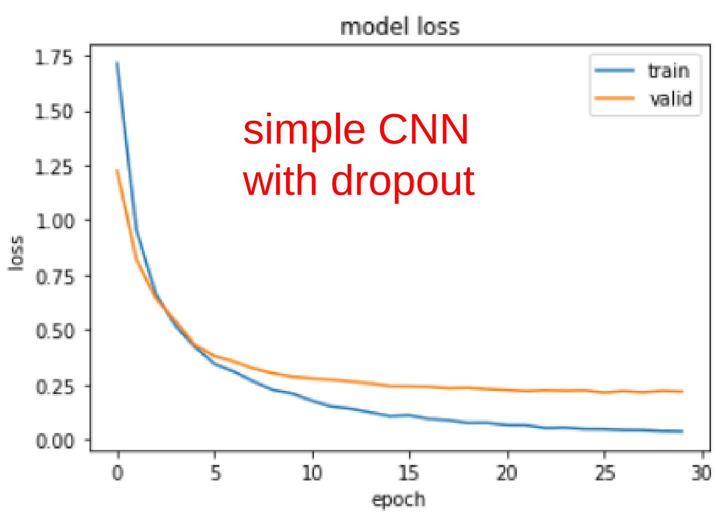
In keras:

```
model.add(layers.GRU(32, dropout=0.2, recurrent_dropout=0.2, input_shape=(None, …)))
```

Dropout can fight overfitting in CNN and recurrent NN







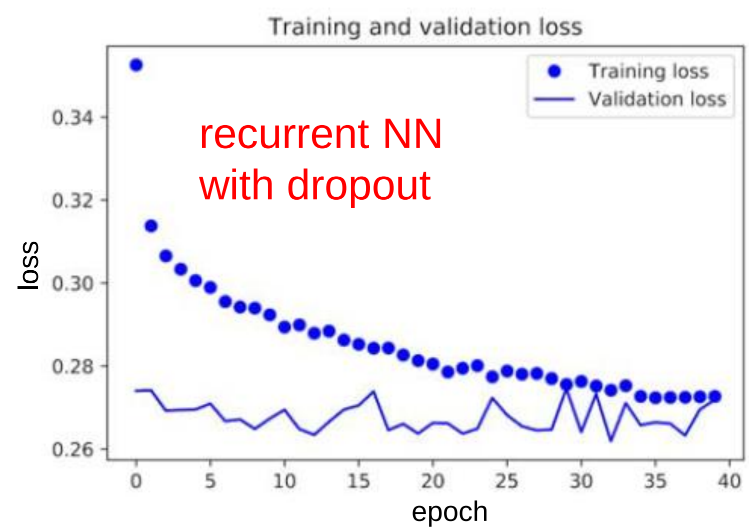
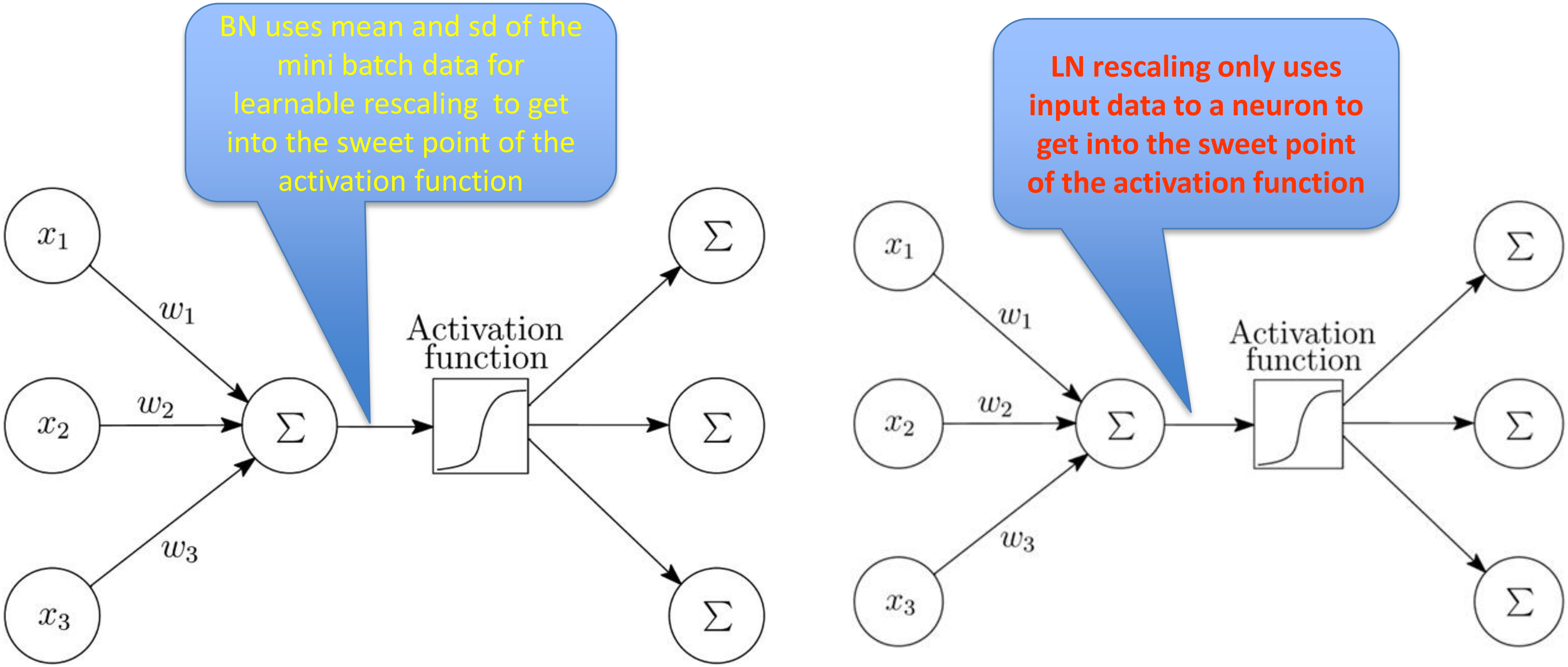


Image credits: F Chollet's book: DL with Python

Batchnormalization is crucial to train deep CNNs
Layernormalization is benefial in RNN: LN $\neq$ BN



Applying BN to RNN would not take into account the recurrent architecture of the NN over which statistics of the input to a neuron might change considerable within the same mini batch. In LN the mean and variance from all of the summed inputs to the neurons in a layer on a single training case are used for normalization .

For details on layer nomarlization see [Ba.Kiros.Hinton2016]

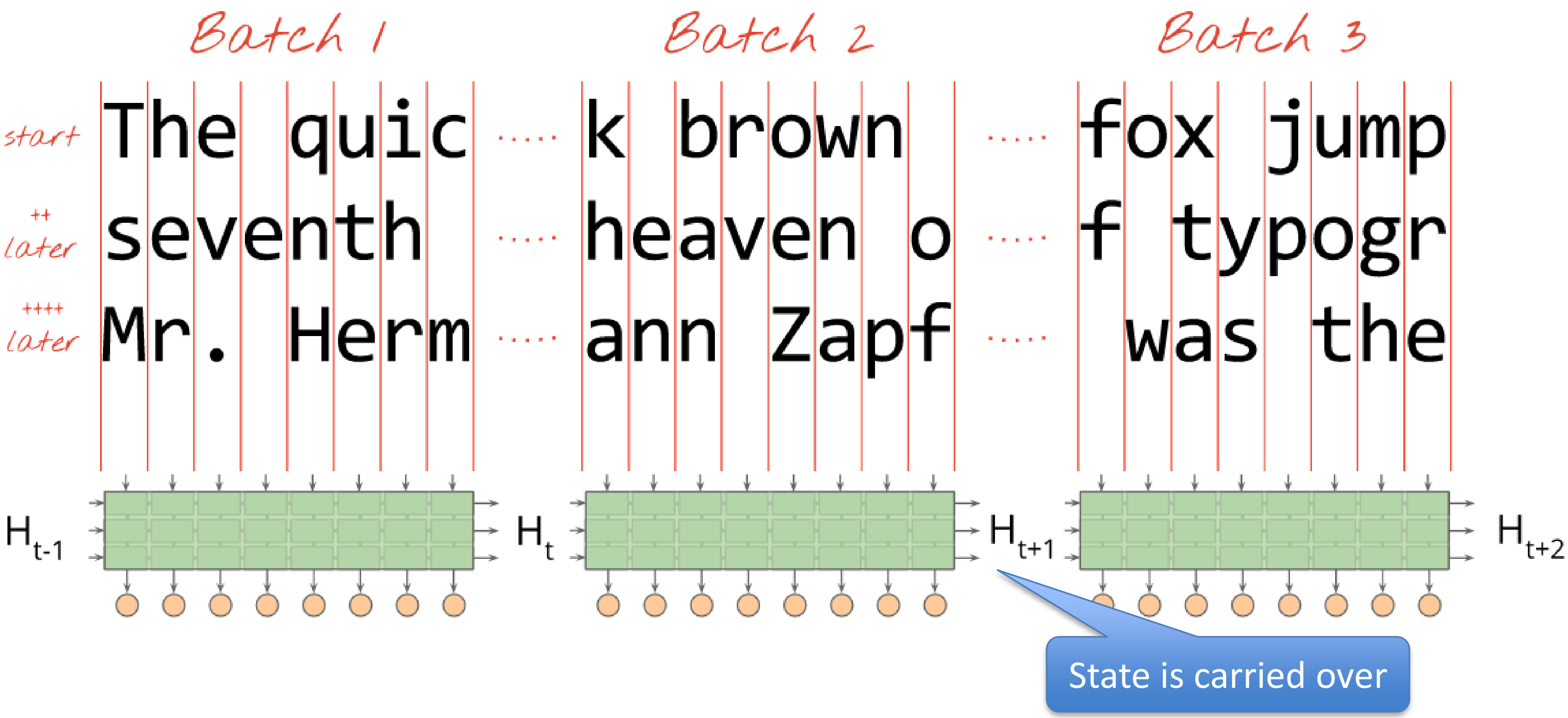
Stateful RNN model

Training a stateful RNNs

- RNN are often trained on sequence data with inherent order
- Sequences are often very long and need to be cut between mini-batches
- By default the hidden state is initialized with zeros in each mini-batch
- In stateful RNN we connect sequences in the right order between mini-batches allowing to make use of the hidden state learned so far
- This requires a careful construction of the mini-batches and an appropriate transfer of the hidden state between mini-batches

Mini-batches in statefull RNN

The gradient is propagated back a fixed amount of steps defined by the size of a mini-batch. In stateful RNNs the hidden state is carried over between mini-batches and hence between connecting sequences given appropriate batches.



Slide credit Martin Gorner (TensorFlow without PhD)

Vanishing/Exploding Gradient problem during training a RNN

Recall: Loss of a mini-batch is used to determine update

mini-batch of size M=8

train data input (S=len(seq)=3):

instance_id	seq_t1	seq_t2	seq_t3
1	$\mathbf{x}_{11}$	$\mathbf{x}_{12}$	$\mathbf{x}_{13}$
2	$\mathbf{x}_{21}$	$\mathbf{x}_{22}$	$\mathbf{x}_{23}$
3	$\mathbf{x}_{31}$	$\mathbf{x}_{32}$	$\mathbf{x}_{33}$
⁞	⁞	⁞	⁞
8	$\mathbf{x}_{81}$	$\mathbf{x}_{82}$	$\mathbf{x}_{83}$

train data target (2 classes, K=2):

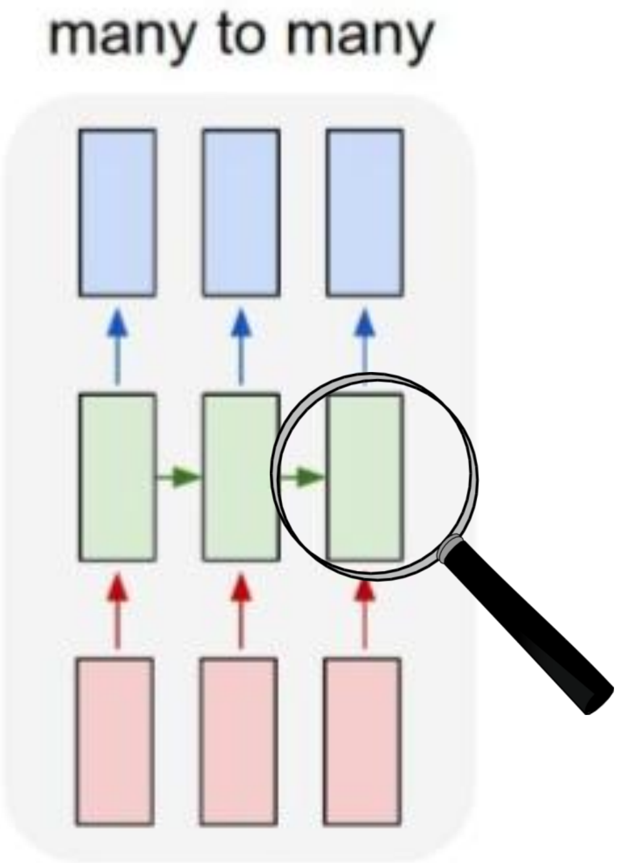
instance_id	y_t1	y_t2	y_t3
1	(1,0)	(1,0)	(0,1)
2	(0,1)	(1,0)	(0,1)
3	(0,1)	(0,1)	-1
⁞	⁞	⁞	⁞
8	(1,0)	(1,0)	(1,0)

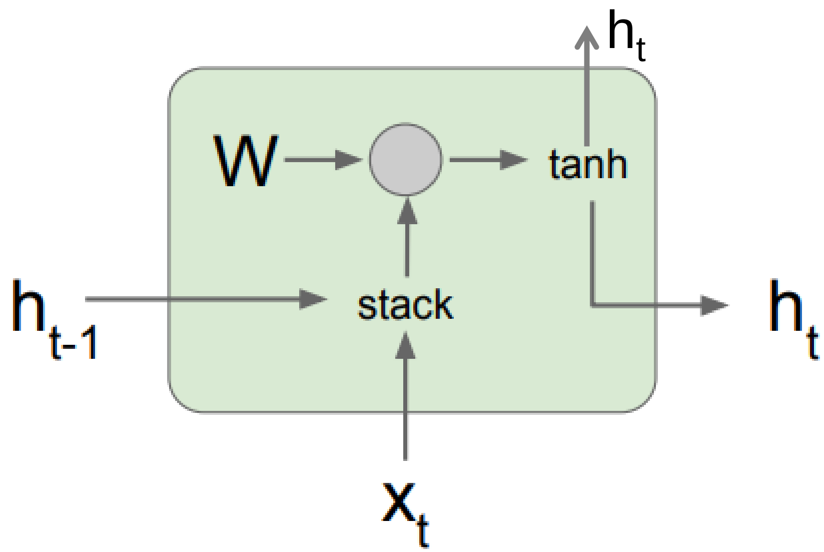
Cost C or Loss is given by the cross-entropy averaged over all instances in mini-batch:

$$\text{Loss} = \frac{1}{8}\sum_{m=1}^{8}\left[\sum_{s=1}^{3}\left(-\sum_{k=1}^{2} y_{\text{msk}} \cdot \log\left(p_{\text{msk}}\right)\right)\right]$$

Based on the mini-batch loss the weights in the tow weight matrices of layer 1 and layer 2 are updated.

Recall: Design of a RNN "cell"

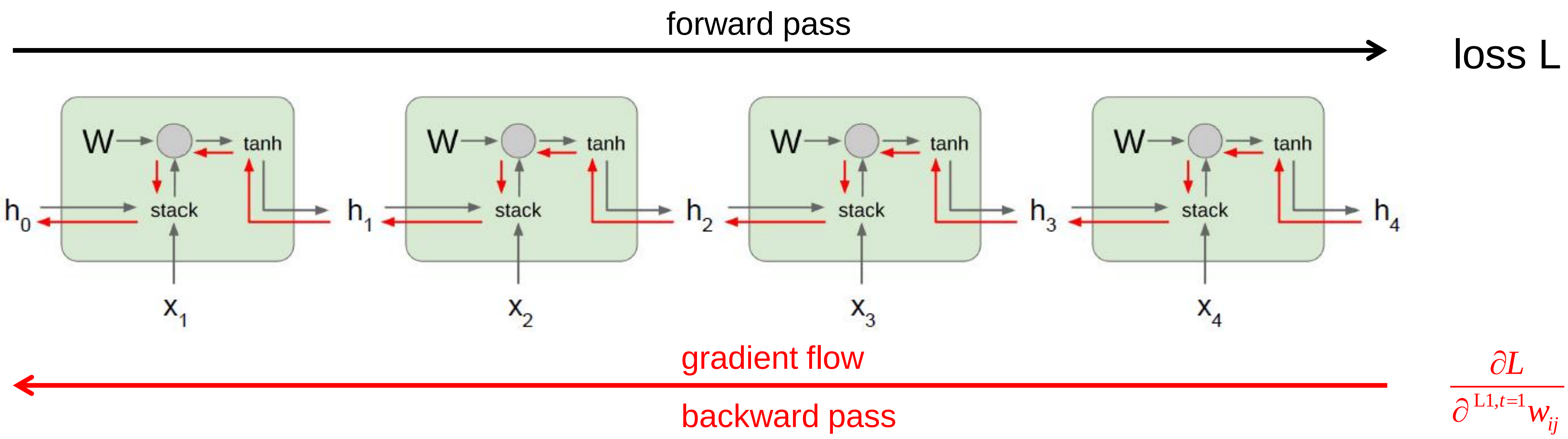




parameter?

$$(|h| + |x|) \cdot |h| + |h$$

Backpropagation in RNNs: Gradient is multiplied at each time step with same factor: Gradient explosion/vanishing



Propagating the gradient of the cost function via chain rule to the first time point involves multiplying at each time step with $\mathbf{W}^T$ (and the derivation of tanh).

⟹ Vanishing gradient if we multiply at each time step with a number <1
(more precisely we have only a number if W is a scalar, otherwise we need to look on the first singular value of $\mathbf{W}^T$)

⟹ Exploding gradient if we multiply at each time step with a number >1
(more precisely we have only a number if W is a scalar, otherwise we need to look on the first singular value of $\mathbf{W}^T$)

Solution: gradient clipping (hack), or use better architecture like LSTM or GRU!

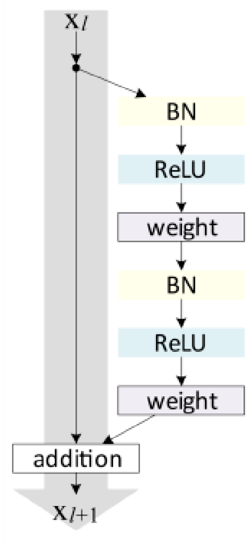
slide (adapted) from cs231 lecture 10

GRU and LSTM cells to avoid vanishing/exploding gradients

Recall: ResNet

- use ResNet like architectures allowing for a gradient highway

(in CNN also batch-normalization and ReLU helped to train deep NN, but cannot naively transfered to recurrent NN)



$$x_{l+1} = x_l + F(x_l)$$

ResNet basic design (VGG-style)

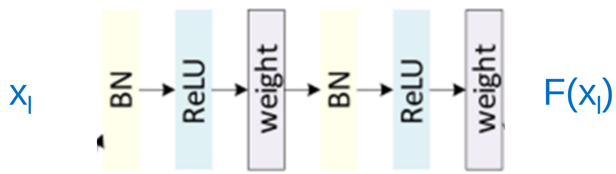
- add shortcut connections every two
- all 3x3 conv (almost)

152 layers:
Why does this train at all?

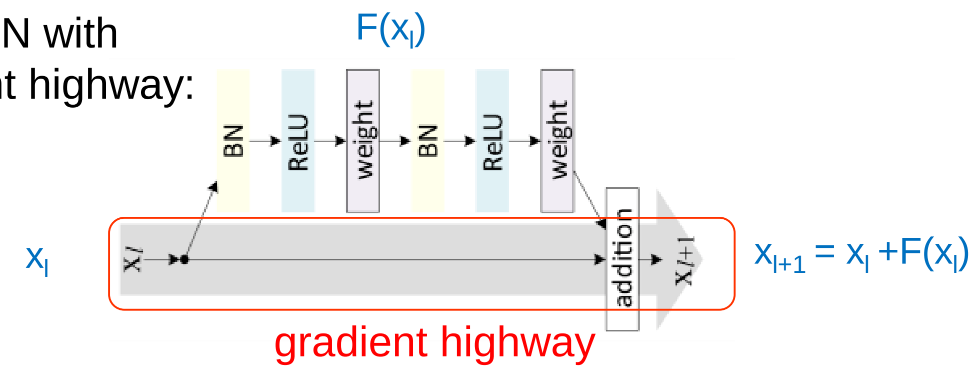
This deep architecture could still be trained, since the gradients can skip layers which diminish the gradient!

Provide gradient highway also in recurrent NN: GRU, LSTM

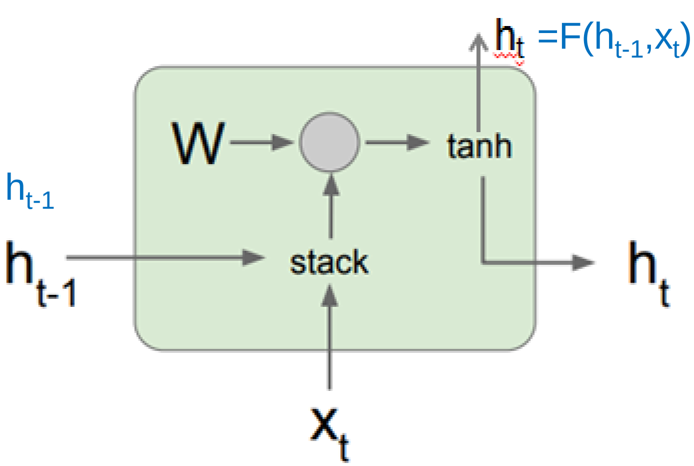
CNN classic:



CNN with gradient highway:

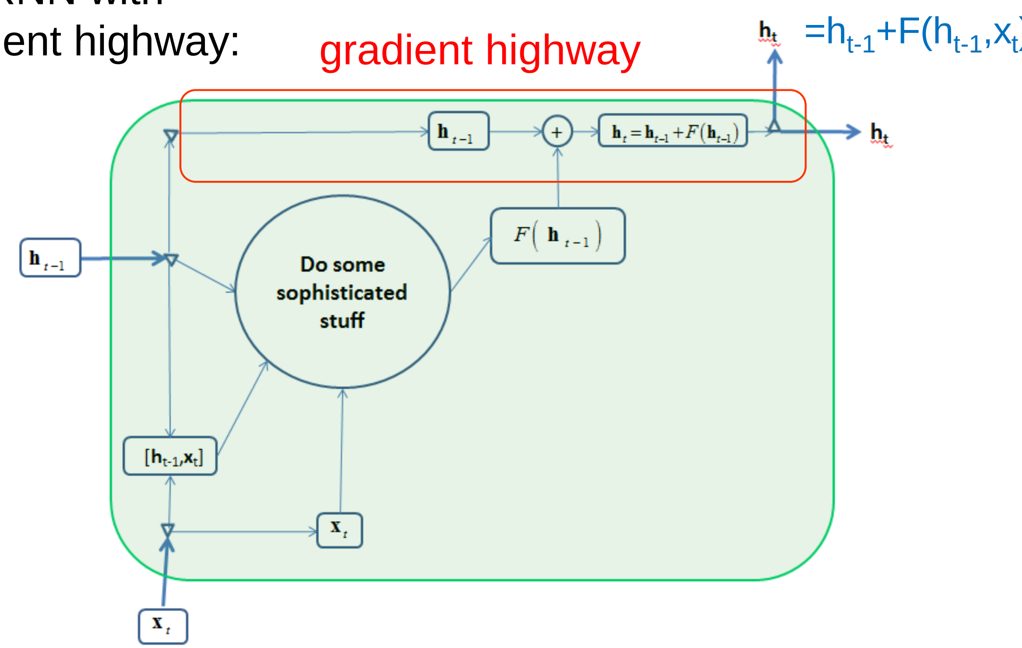


RNN classic:

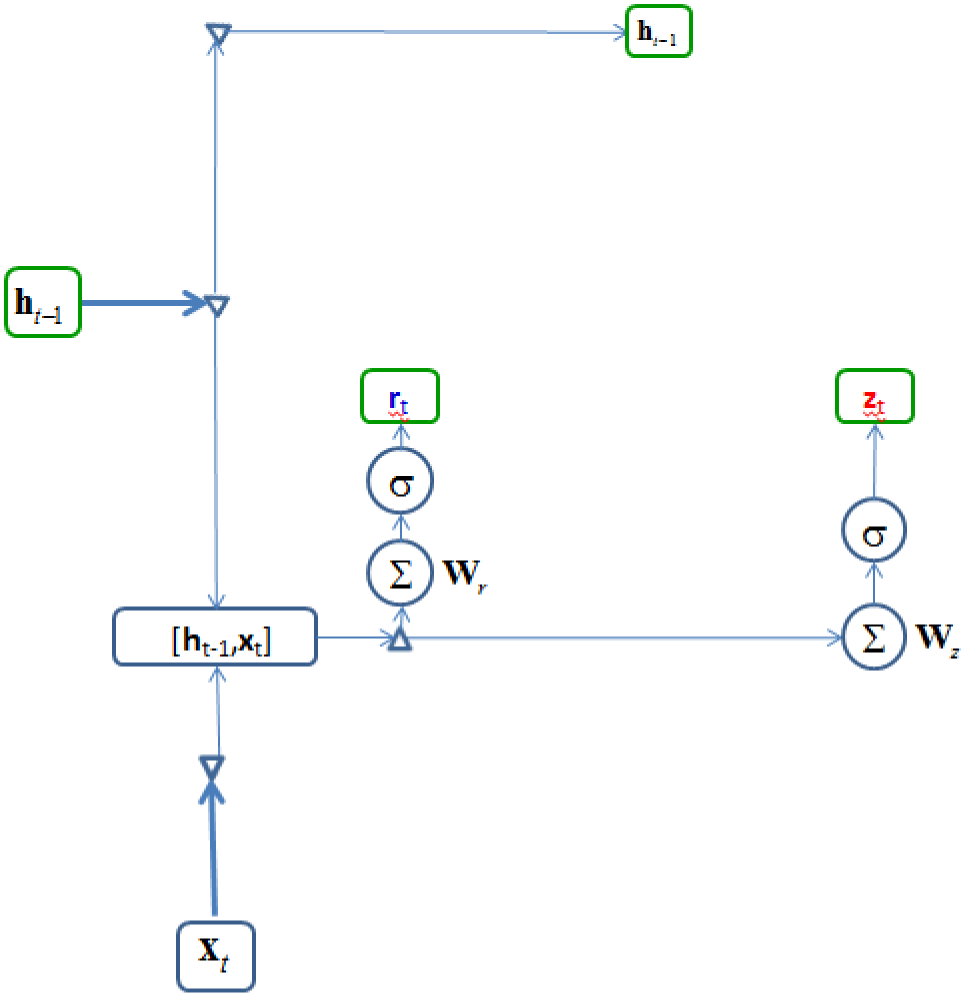


$$\mathbf{h}_t = \tanh\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W} + \mathbf{b}\right)$$

RNN with gradient highway:



Towards Gated Recurrent Units (GRU)



$$\mathbf{r}_t = \text{gate}_{r,t} = \text{sigmoid}\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_r + \mathbf{b}_r\right)$$

$$\mathbf{z}_t = \text{gate}_{\text{update}} = \text{sigmoid}\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_z + \mathbf{b}_z\right)$$

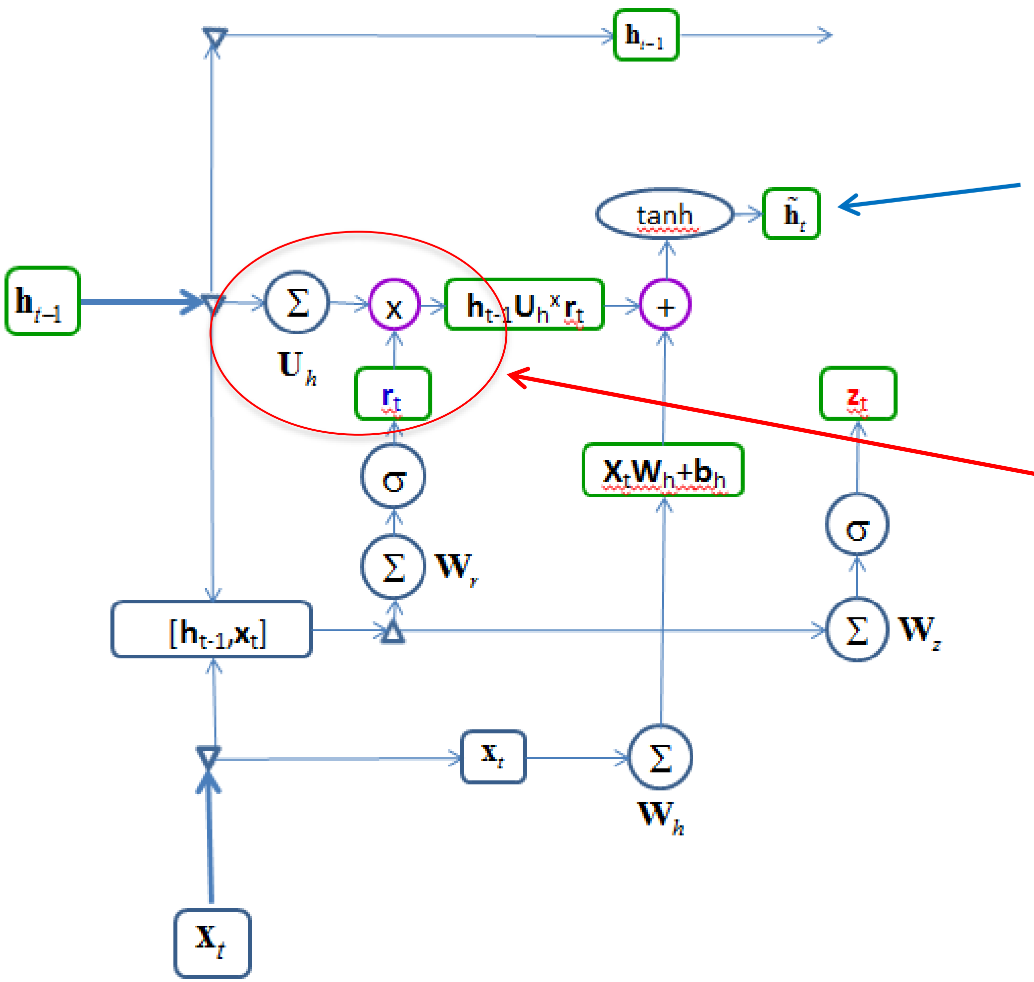
Idea:

The **relevant gate r** controls which part of the previous hidden state is relevant for making a prediction or should be dropped

The **updated gate z** controls how much information from the previous hidden layer h_{i-1} and the new input should be propagated to the current hidden layer h_i.

Remark: all internal vectors/tensors with green frame have same length/shape.

Towards Gated Recurrent Units (GRU)



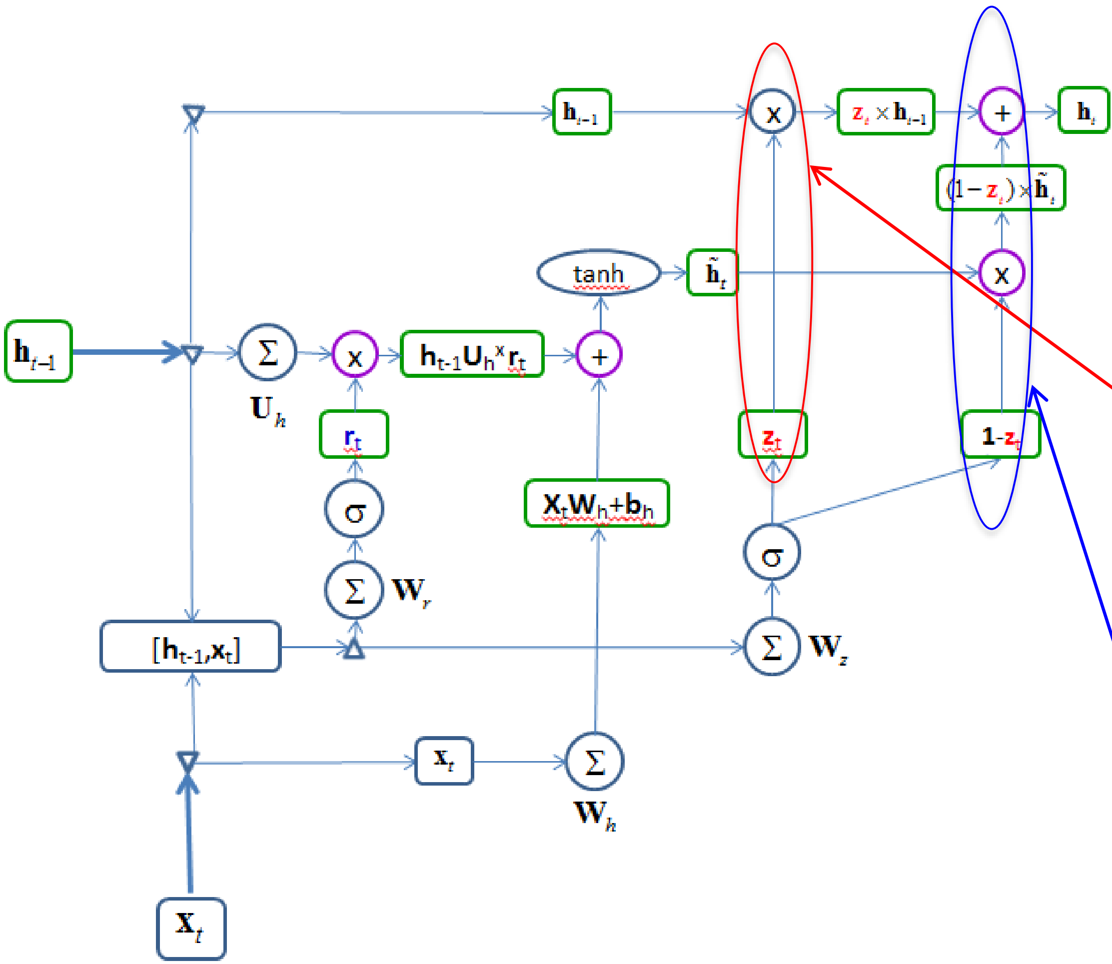
The new proposed state $\tilde{h}$ is:

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{x}_t \cdot \mathbf{W}_h + \mathbf{b}_h + \underline{\mathbf{h}_{t-1}\mathbf{U}_h \otimes \mathbf{r}_t}\right)$$

Here, we use the relevant gate r to control what part of $\boldsymbol{h}_{t-1}$ we need to compute a new proposal.

Remark: all internal vectors with green frame have same length.
All ops within a purple circle are performed per element-wise on the ingoing vectors

The Gated Recurrent Unit (GRU)



The new hidden state is:

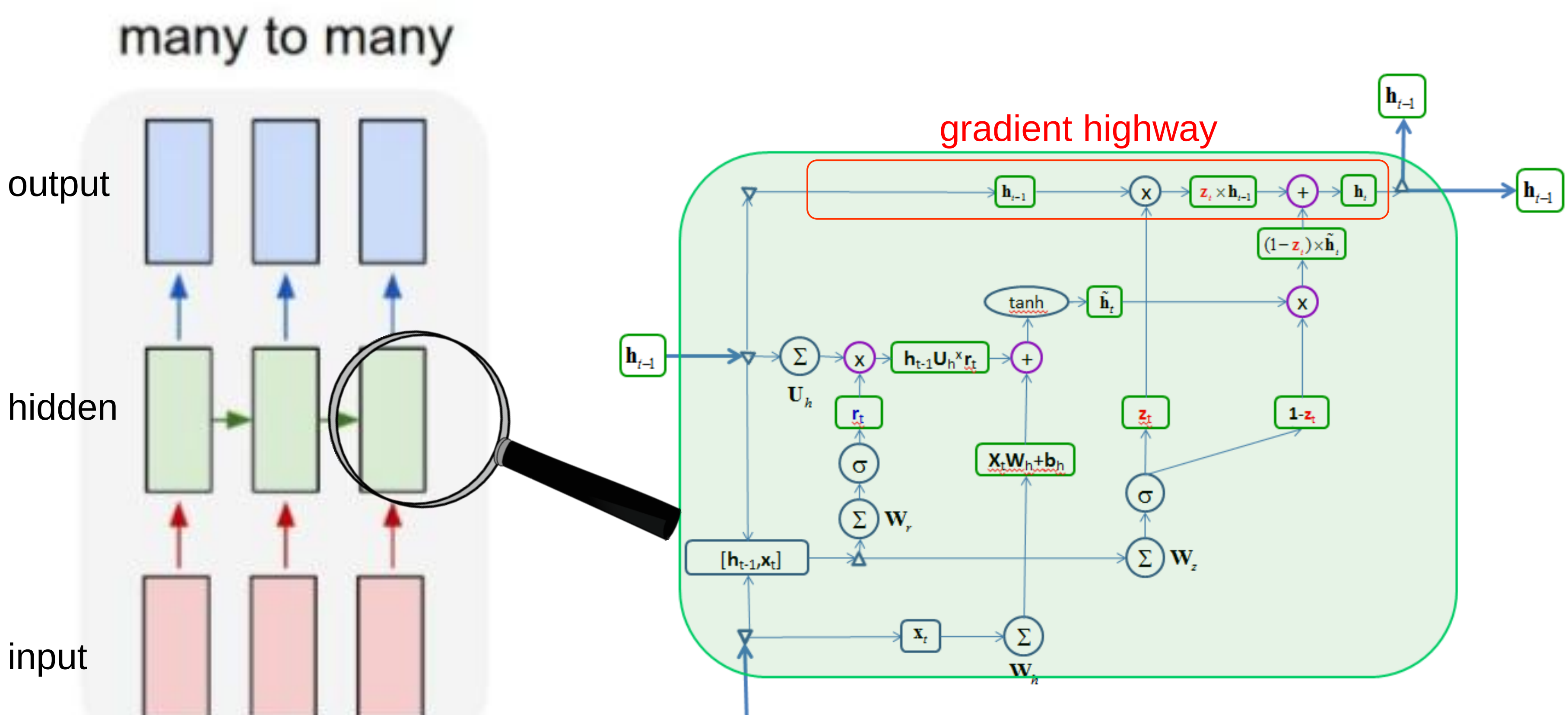
$$h_t = (1\text{-}z_t) \otimes \tilde{h}_t \oplus z_t \otimes h_{t-1}$$

If all elements of z_t are 1 then the hidden state stays unchanged.

The updated gate z_t controls how much of the previous hidden state h_{t-1} and the new input x_t should be propagated to the current hidden state h_t

The updated gate z_t controls also how much information from proposed new state $\tilde{h}$ is entering the new state

Solution via “highway allowing” architecture: GRU

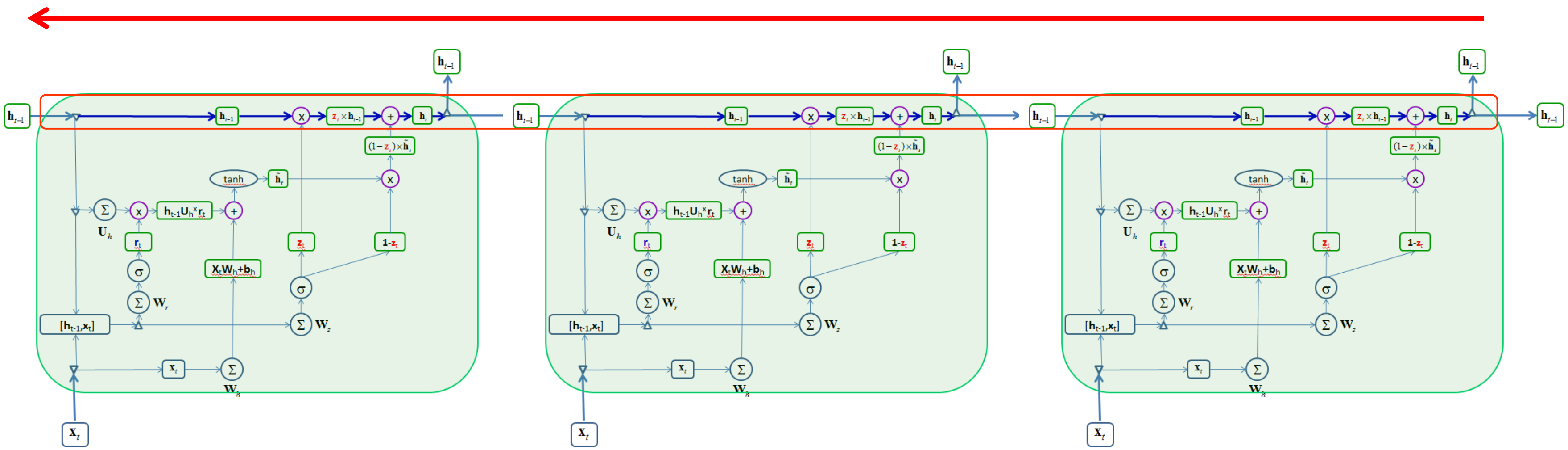


Remark: all internal vectors with green frame have same length.
All ops within a purple circle are performed per element-wise.

The gradient high-way avoids gradient vanishing. The GRU also avoids gradient explosion since the element-wise operations on vector-elements that change over the time steps, avoids multiplying the gradients with the same number in each step.

The Gated Recurrent Unit (GRU): Gradient Flow

Uninterrupted gradient flow!



parameter?

Relevant gate: $\mathbf{r}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_r + \mathbf{b}_r\right)$

Update gate: $\mathbf{z}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_z + \mathbf{b}_z\right)$

Proposed hidden state: $\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{x}_t \cdot \mathbf{W}_h + \mathbf{b}_h + \mathbf{h}_{t-1}\mathbf{U}_h \otimes \mathbf{r}_t\right)$

New hidden state is: $\mathbf{h}_t = (\mathbf{1}\text{-}\mathbf{z}_t) \otimes \tilde{\mathbf{h}}_t \oplus \mathbf{z}_t \otimes \mathbf{h}_{t-1}$

$$(|h| + |x|) \cdot |h| + |h$$

$$+(|h| + |x|) \cdot |h| + |h|$$

$$+ \ (|x|) \cdot |h| + |h| + \ (|h|) \cdot |h|$$

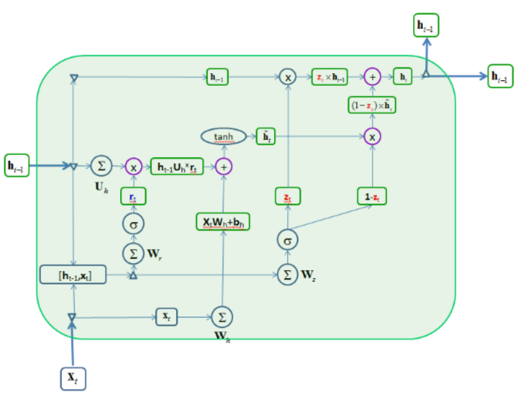
$$= 3 \cdot [(|h| + |x|) \cdot |h| + |h|]$$

A simplified variation, the Gated Recurrent Unit, or GRU, introduced by Cho, et al. (2014).

GRU in keras

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

length of internal state

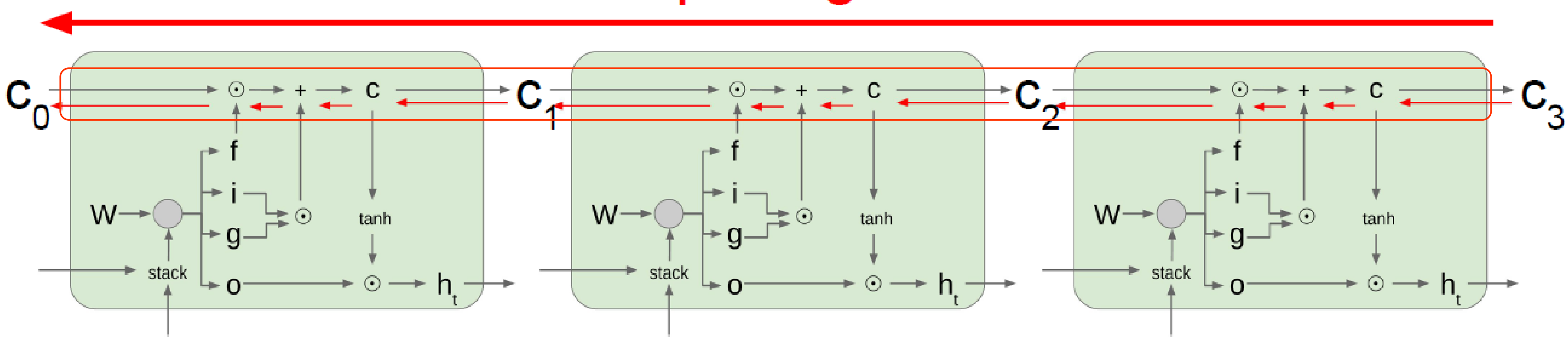


All internal vectors within GRU green frame have same length.
All ops within a purple circle are performed per element-wise on the ingoing vectors

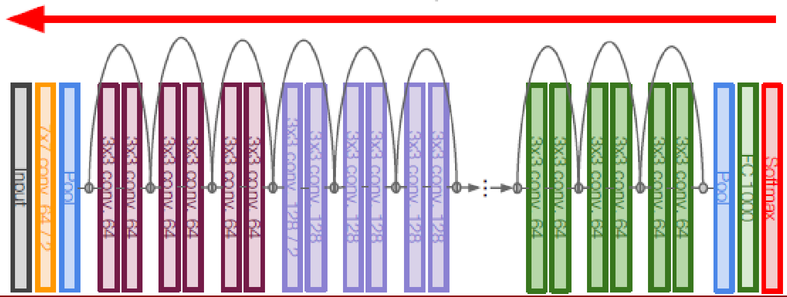
Long Short Term Memory (LSTM): Gradient Flow

LSTM has an additional cell state C for a “long term memory”.

Uninterrupted gradient flow!



Similar to ResNet!

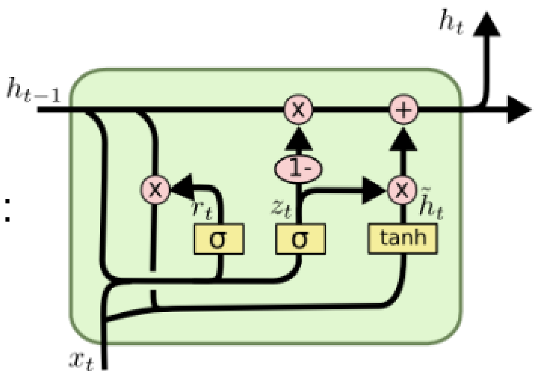


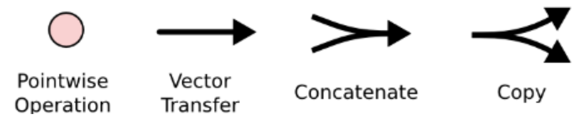
Long Short Term Memory networks (LSTM) were introduced by [Hochreiter & Schmidhuber (1997).]

Slide credit (modified): cs231 2017 stanford

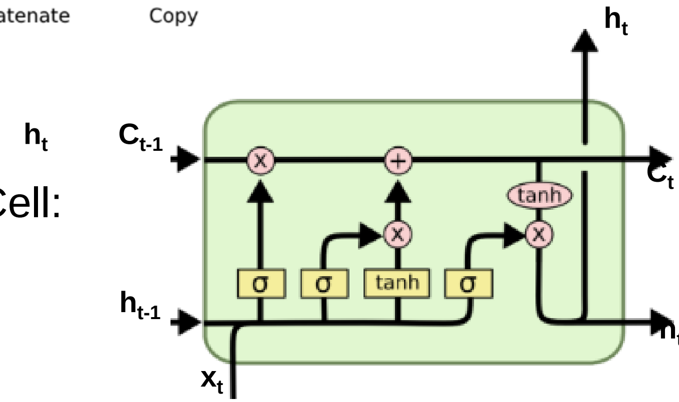
Long Short Term Memory cell (LSTM) as GRU-extension

GRU-Cell:





LSTM-Cell:



2 gates, 1 cell states (h)

Relevant gate: $\mathbf{r}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_r + \mathbf{b}_r\right)$

Update gate: $\mathbf{z}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_z + \mathbf{b}_z\right)$

Proposed hidden state: $\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{x}_t \cdot \mathbf{W}_h + \mathbf{b}_h + \mathbf{h}_{t-1}\mathbf{U}_h \otimes \mathbf{r}_t\right)$

New hidden state is: $\mathbf{h}_t = (\mathbf{1}\text{-}\mathbf{z}_t) \otimes \tilde{\mathbf{h}}_t \oplus \mathbf{z}_t \otimes \mathbf{h}_{t-1}$

3 gates, 2 cell states (S:h, L:C)

Forget gate: $\mathbf{f}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_f + \mathbf{b}_f\right)$

Input gate: $\mathbf{i}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_i + \mathbf{b}_i\right)$

Output gate: $\mathbf{o}_t = \sigma\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_o + \mathbf{b}_o\right)$

Proposed cell state: $\tilde{\mathbf{C}}_t = \tanh\left([\mathbf{h}_{t-1}, \mathbf{x}_t] \cdot \mathbf{W}_C + \mathbf{b}_C\right)$

New L cell state: $\mathbf{C}_t = \mathbf{f}_t \otimes \mathbf{C}_{t-1} \oplus \mathbf{i}_t \otimes \tilde{\mathbf{C}}_t$

New S hidden state: $\mathbf{h}_t = \mathbf{o}_t \otimes \tanh\left(\mathbf{C}_t\right)$

Credits: Colah blog http://colah.github.io/posts/2015-08-Understanding-LSTMs
LSTM: A Search Space Odyssey https://arxiv.org/abs/1503.04069 "all variants of LSTM achieved comparable performance"
An Empirical Exploration of Recurrent Network Architectures, Jozefowicz et al., 2015 10k RNN architectures – few outperformed lstm on few tasks

Long Short Term Memory (LSTM) in keras

dimension of vocabulary

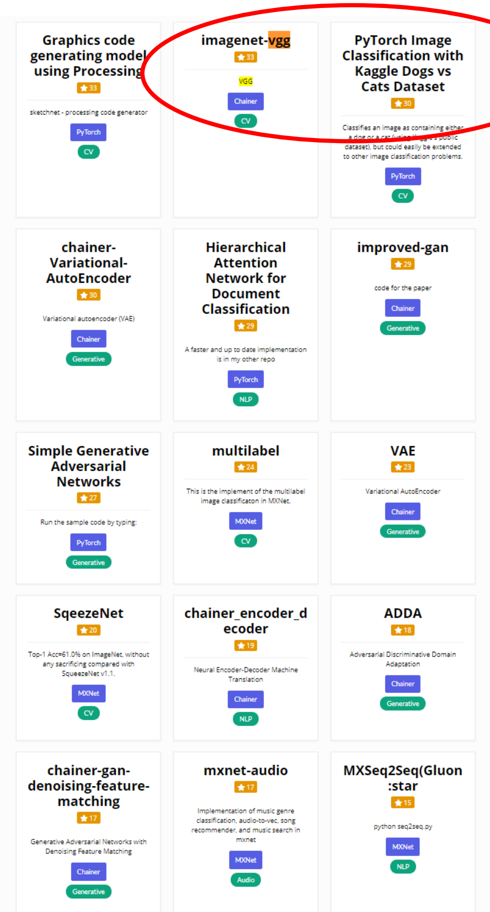
dimension of embedding

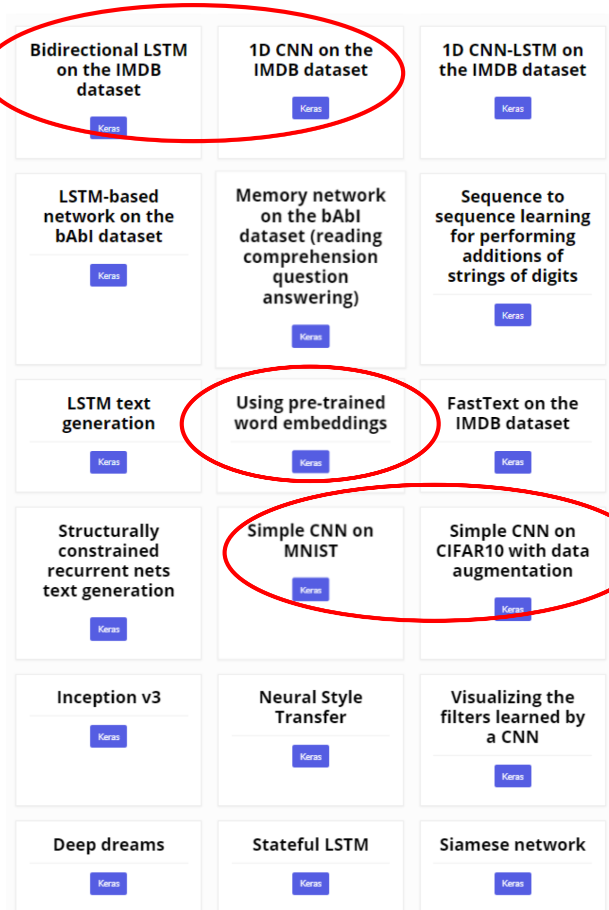
```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

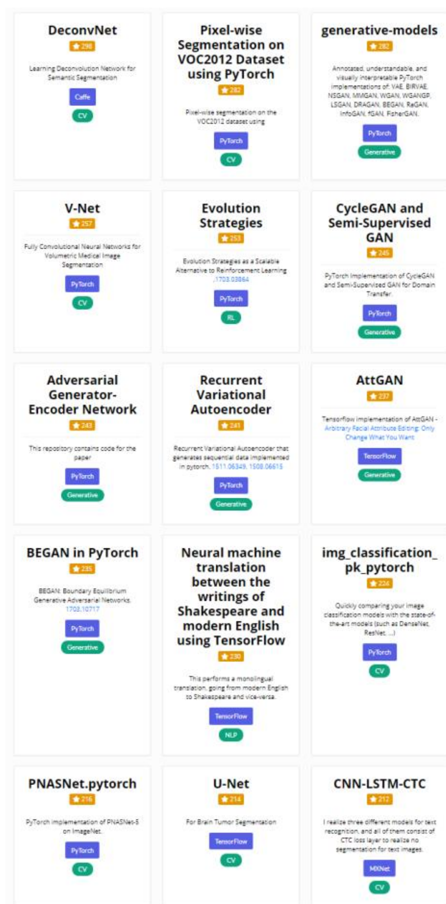
Model zoo: many pretrained NN are out there

https://modelzoo.co/

Base pretrained models and datasets in pytorch (MNIST, SVHN, CIFAR10, CIFAR100, STL10, AlexNet, VGG16, VGG19, ResNet, Inception, SqueezeNet)

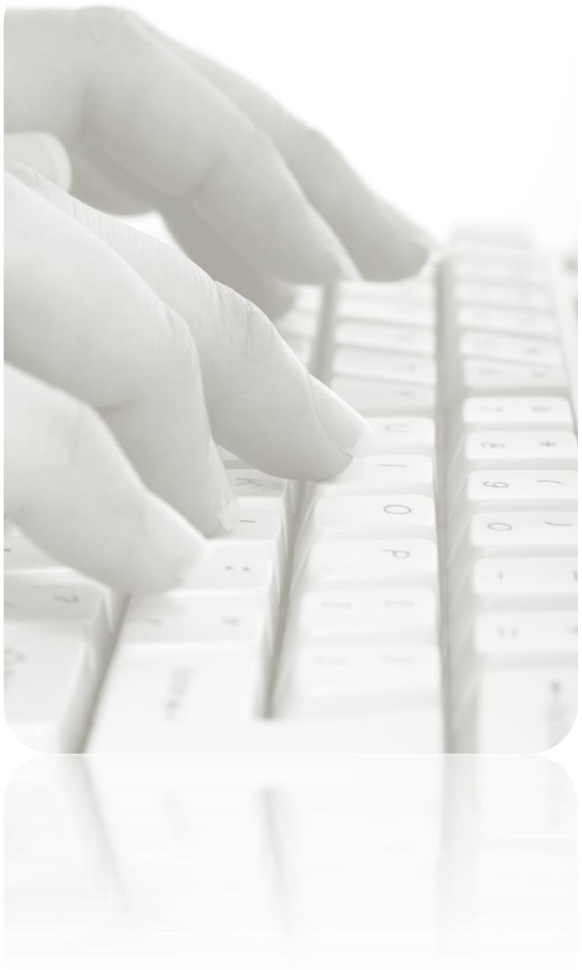






Can LSTM improve your conv1D series predictions?

- Work through the instructions in the second exercise in day 6 using https://github.com/tensorchiefs/dl_course_2018/blob/master/notebooks/12_LSTM_vs_1DConv.ipynb

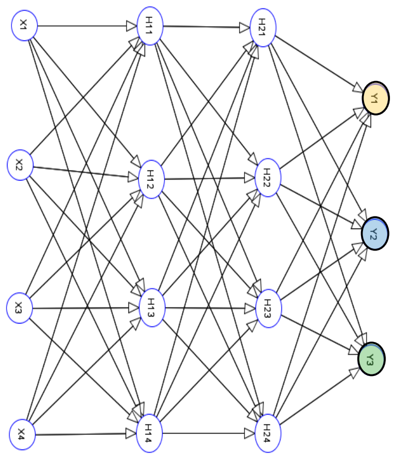


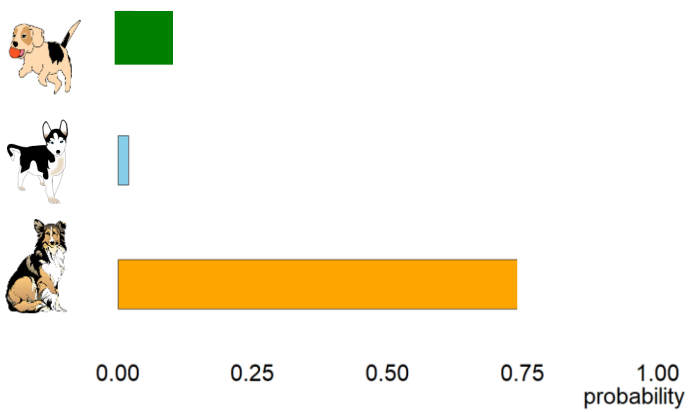
NN can predict class-labels and numeric values, but…

How sure is the NN?

What do we get from a Deep Learning model?

Suppose you train a classifier to discriminate between 3 dog breeds





The prediction is “collie” because it gets the highest probability: p_{max}=0.75

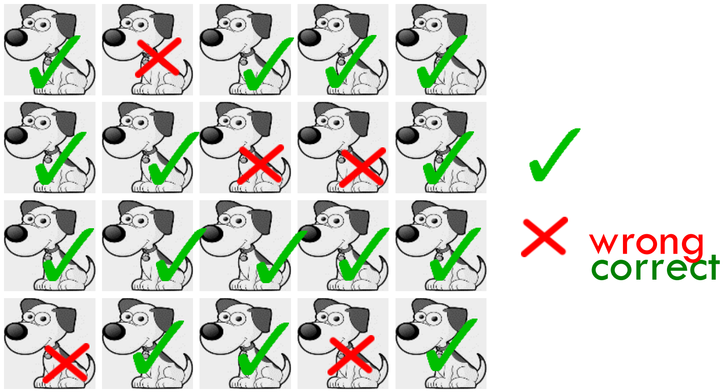
What is the probability telling?

p_{max}=0.75

Among many predications that had p_{max}=0.75, we expect that on average 75% of these predictions are correct and only 25% predictions are wrong

→ Then the classifier produces *calibrated probabilities*

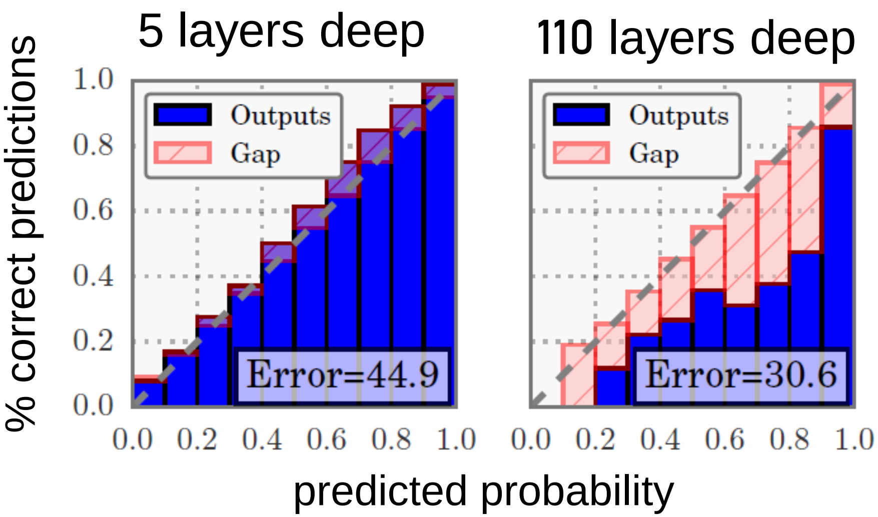
Sample of images where the predicted class got p=0.75:



Do CNNs produce calibrated probabilities?

Guo et al. (2017)
[On Calibration of Modern NN]

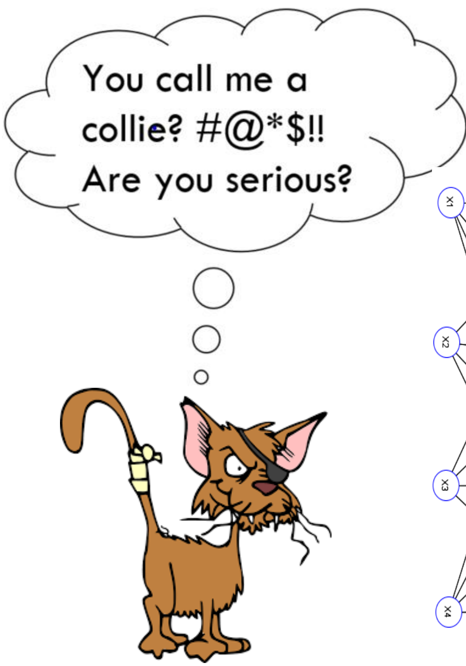
The deeper CNNs get
- the fewer miss-classifications
- the less well calibrated they get

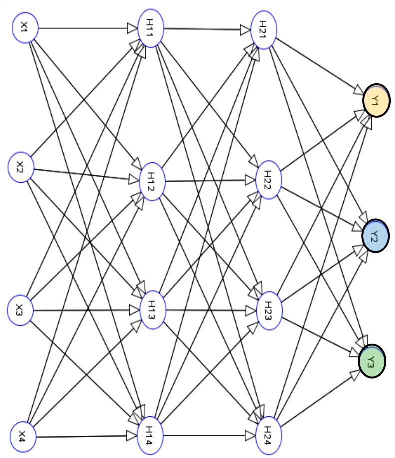


Good news:
deep NN can be “recalibrated” and then we get calibrated probabilities.

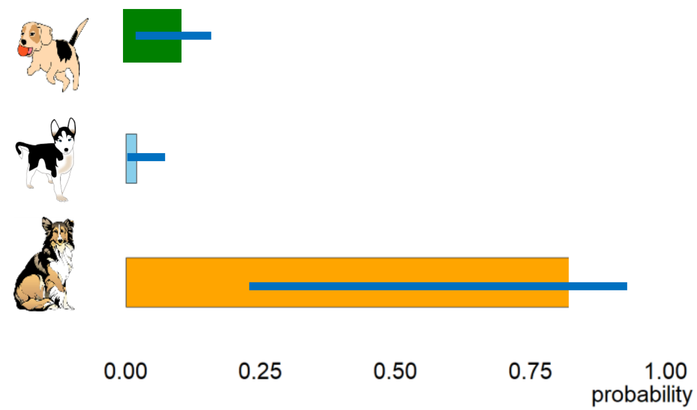
CNNs yield high accuracy and calibrated probabilities, but…

What happens if a novel class is presented to the CNN?



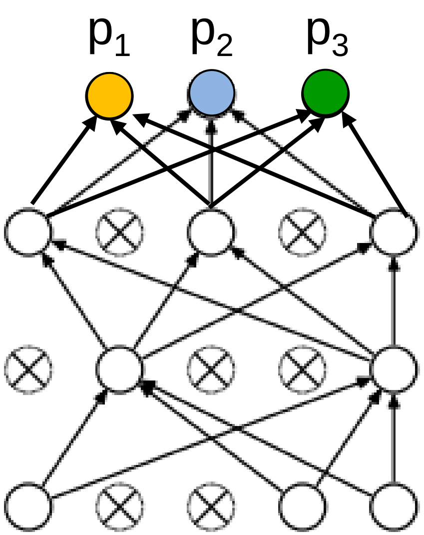


Plain wrong !



We need some error bars!

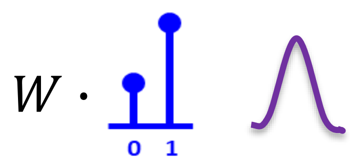
MC Dropout and Bayesian Neural Networks

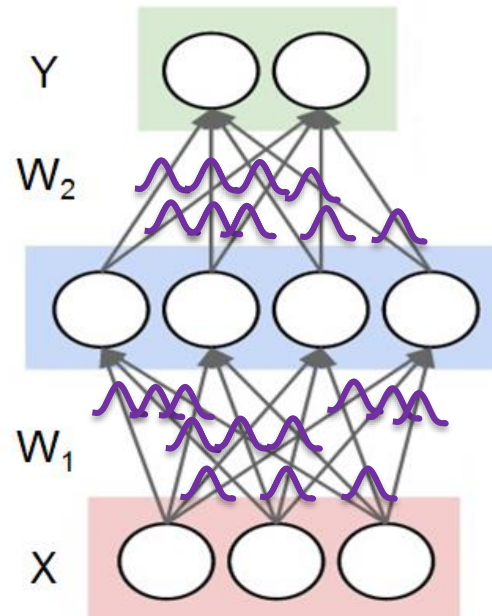


MC Dropout
Randomly drop nodes in each run
→ Ususally done during training

Dropout in test time

Yarin Gal* (2015): we learn a whole weight distribution



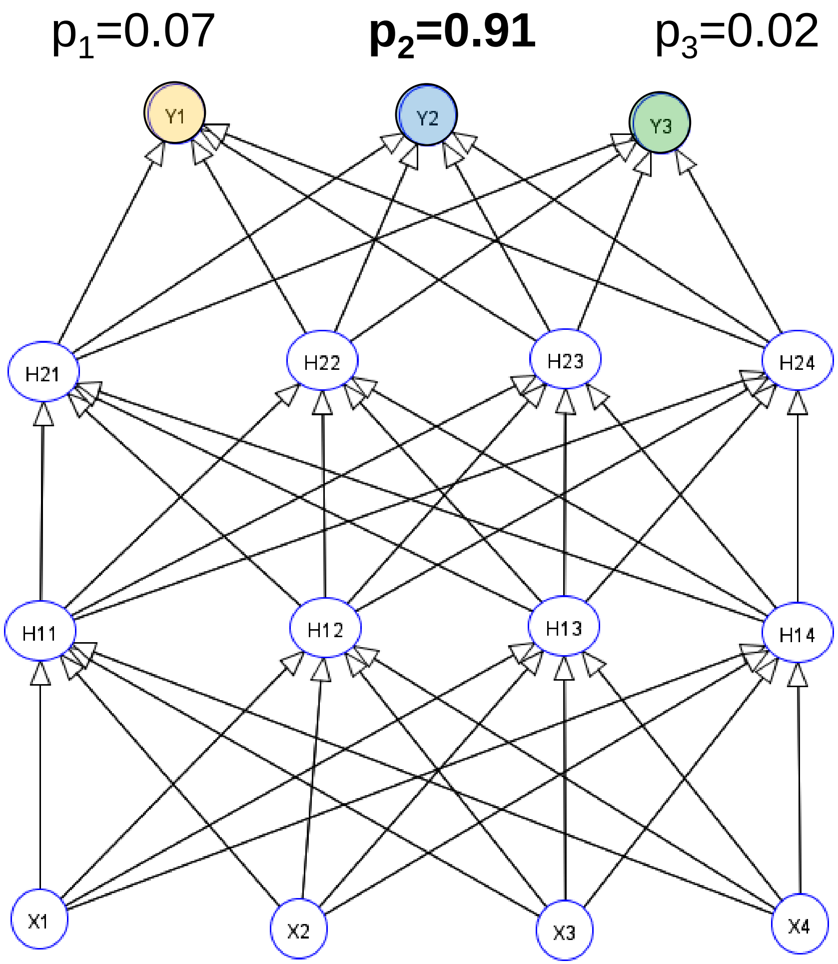


Bayesian NN

→ We should sample from weight distribution during test time

*Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning https://arxiv.org/abs/1506.02142

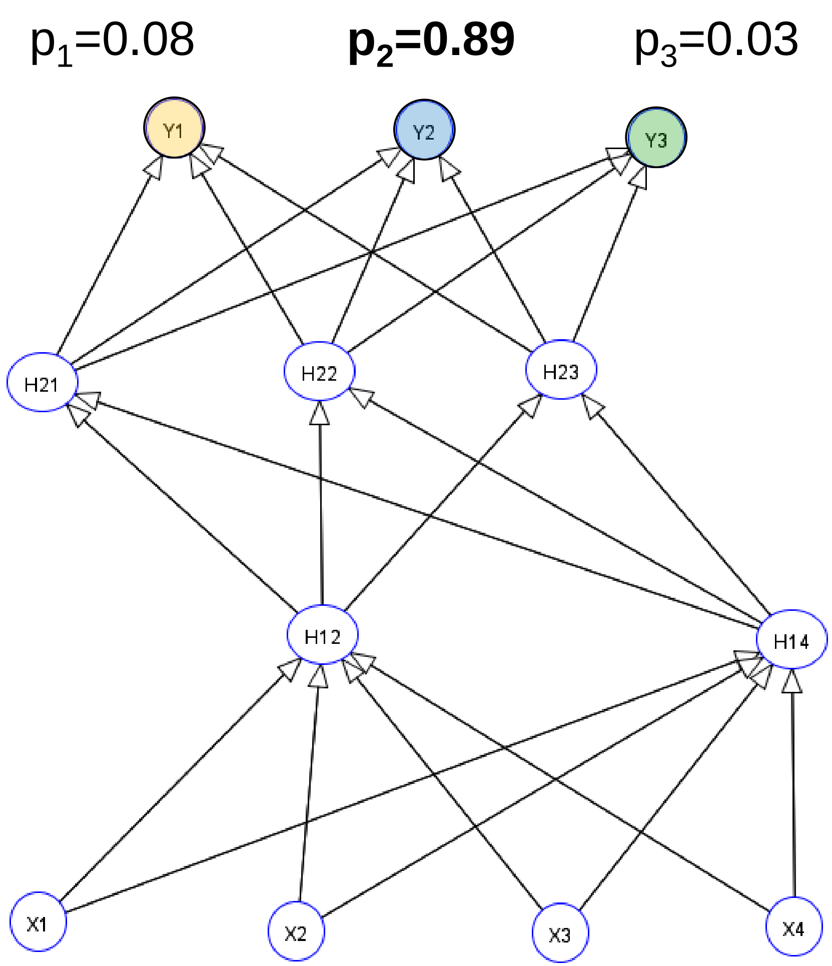
No MC Dropout

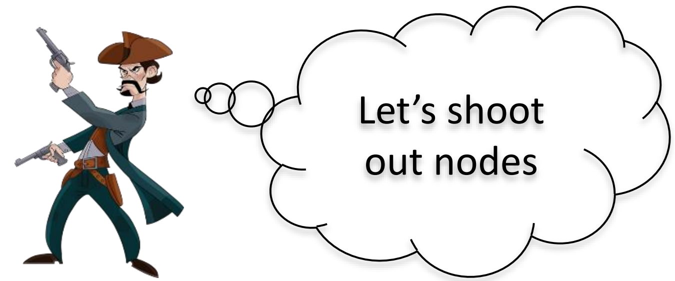


Probability of predicted class: $\mathbf{p_{max}}$

Input: image pixel values

Use dropout at test time: Run 1

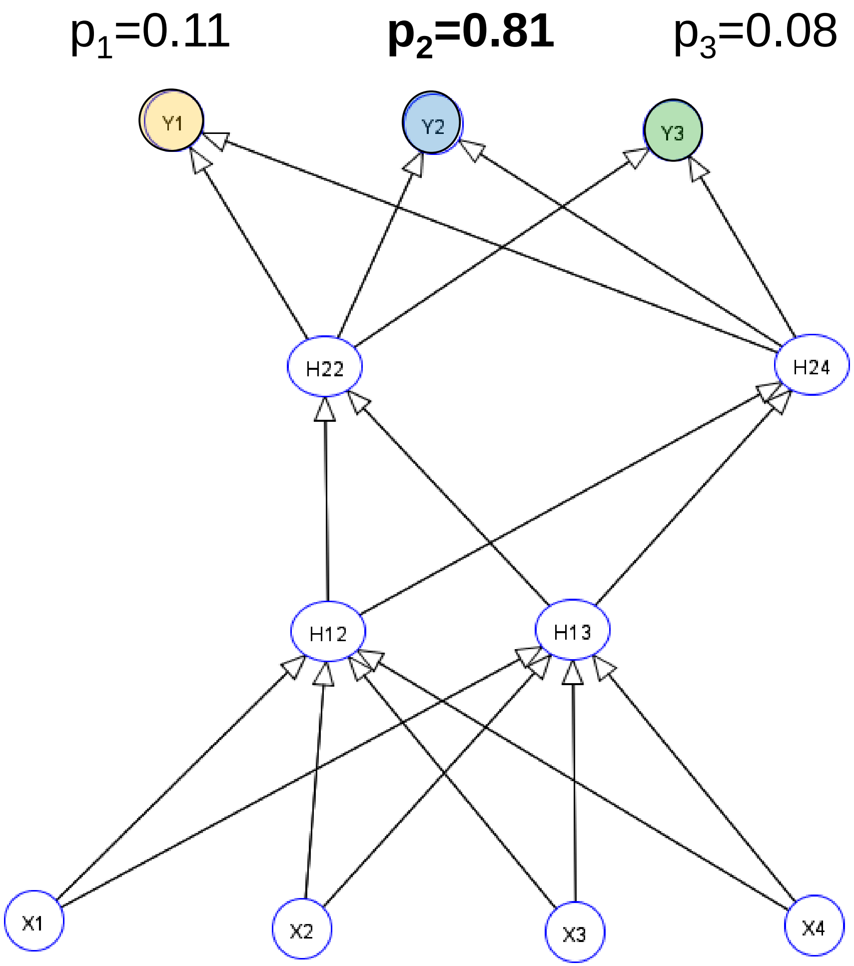


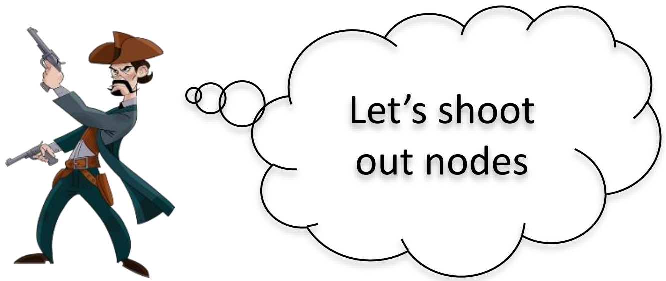


Stochastic dropout of units

Same input image

Use dropout at test time: Run 2

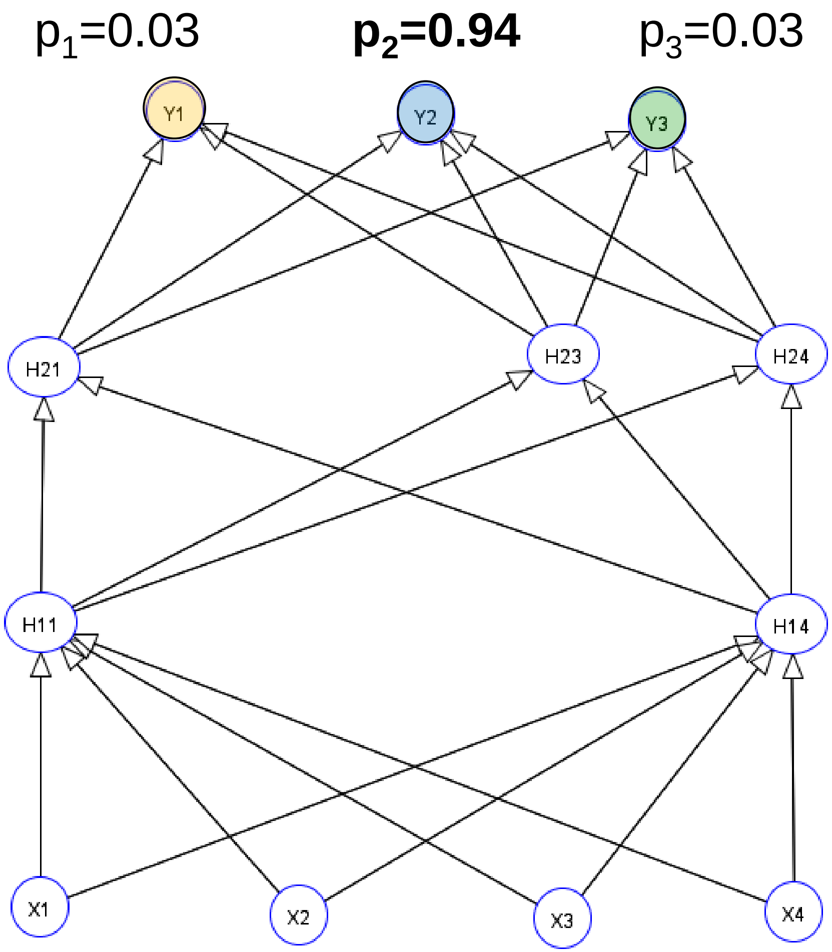


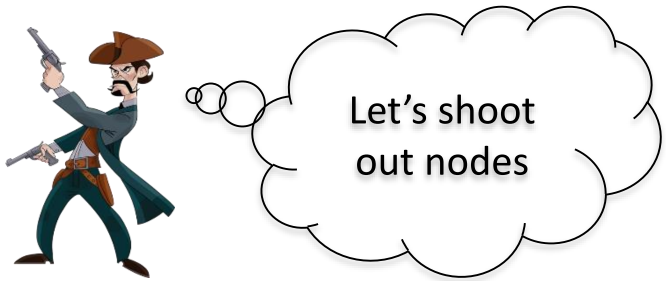


Stochastic dropout of units

Same input image

Use dropout at test time: Run 3

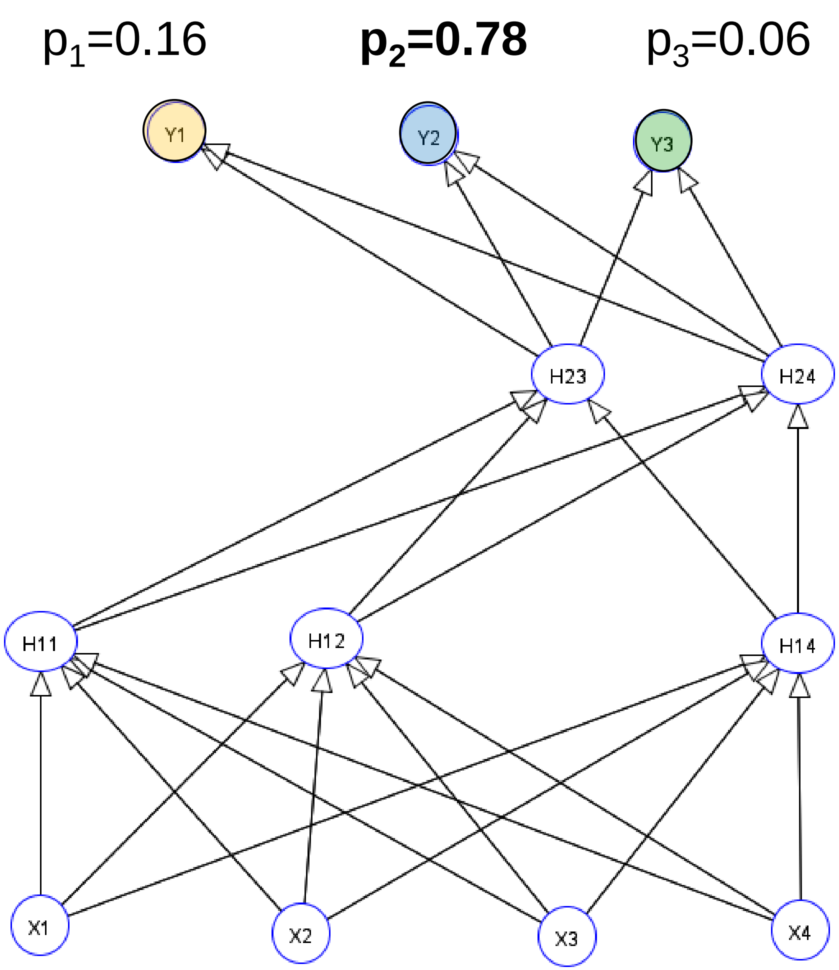


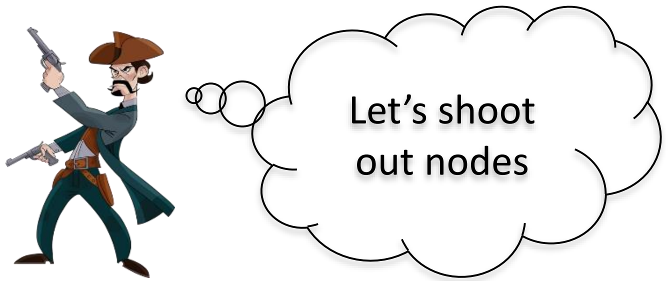


Stochastic dropout of units

Same input image

Use dropout at test time: Run 4



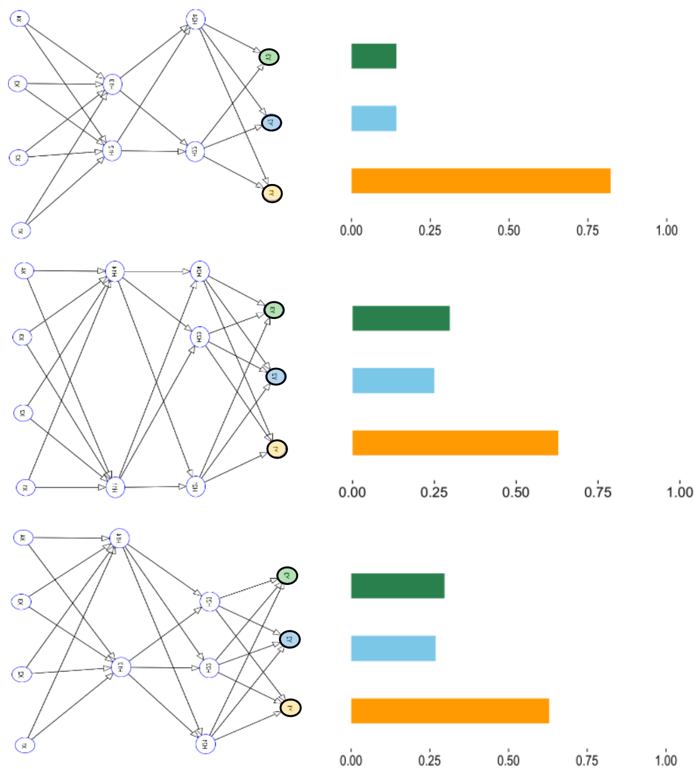


Stochastic dropout of units

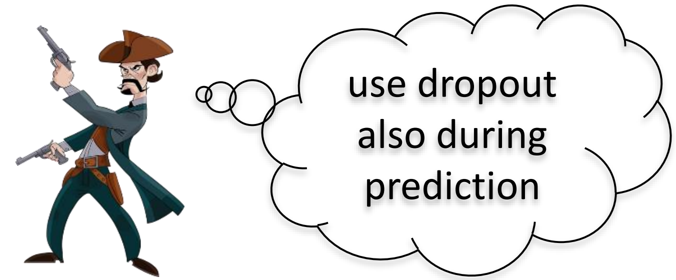
Same input image

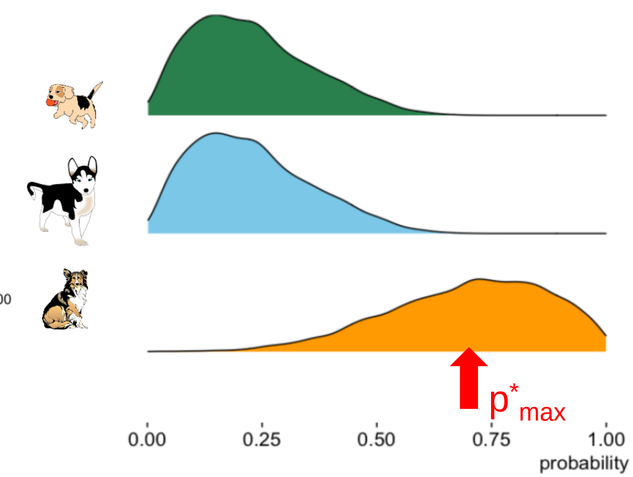
MC probability prediction

Many Dropout Runs in forward pass



...





CNN predicts class "collie"
but with high uncertainty

Remark: Mean of marginal give components of mean in multivariate distribution.

What to get from the MC* probability distribution

The center of mass **quatifies the** predicted probablilty

p^*_{max}

The spread **quantifies in addition the** uncertainty **of the predicted probability**

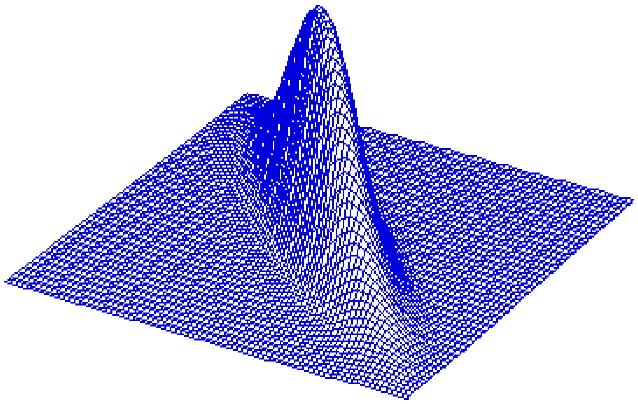
σ^* total standard deviation

PE^* entropy,

MI^* mutual information

VR* variation ratio

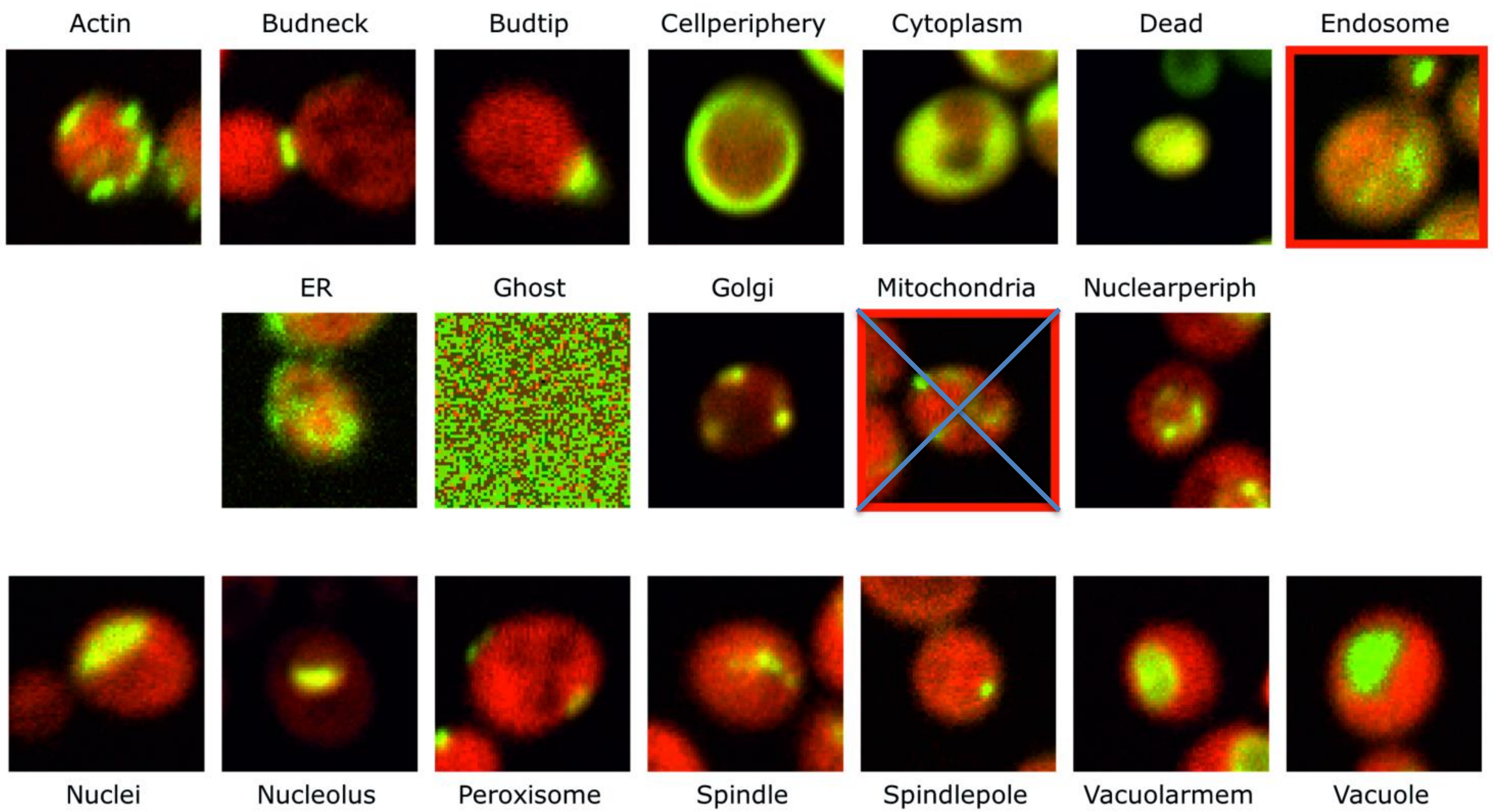
f* vote ratio



Remark: to go from image to patient level we use the average of all measures as baseline

Evaluate method on some real data

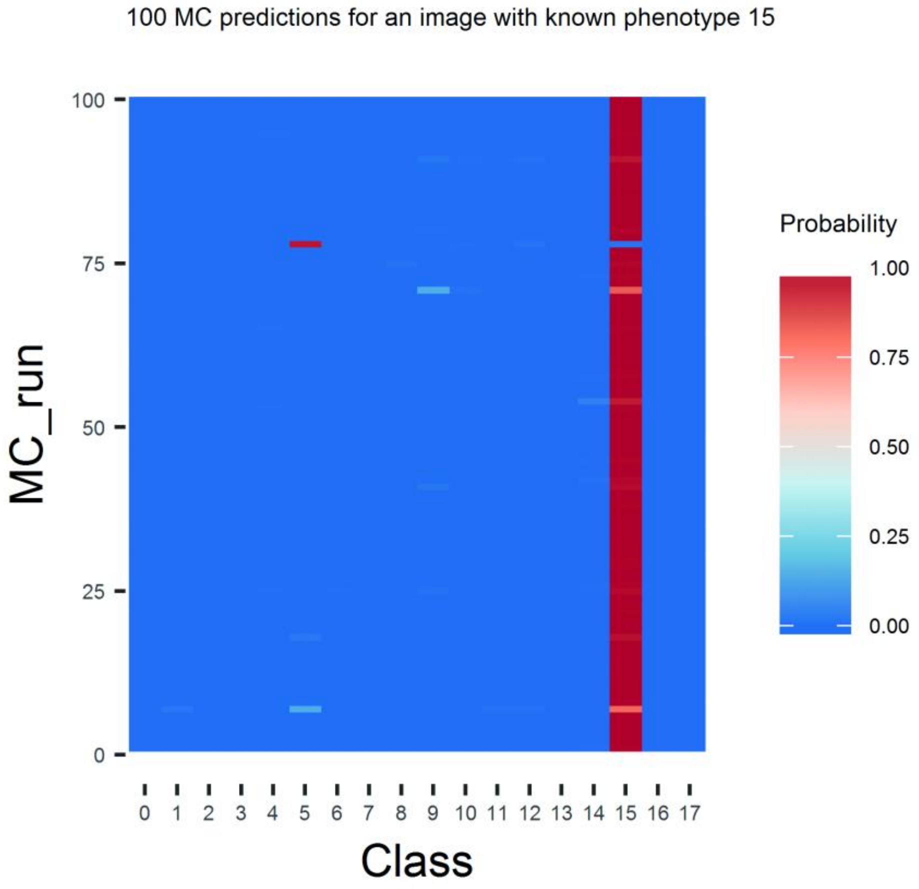
Experiment with unknown phenotype



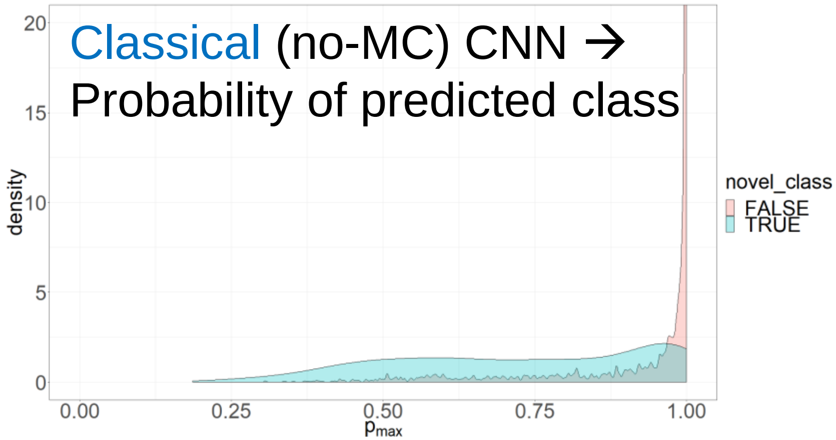
Dürr O, Murina E, Siegismund D, Tolkachev V, Steigele S, Sick B. Know when you don't know, Assay Drug Dev Technol.

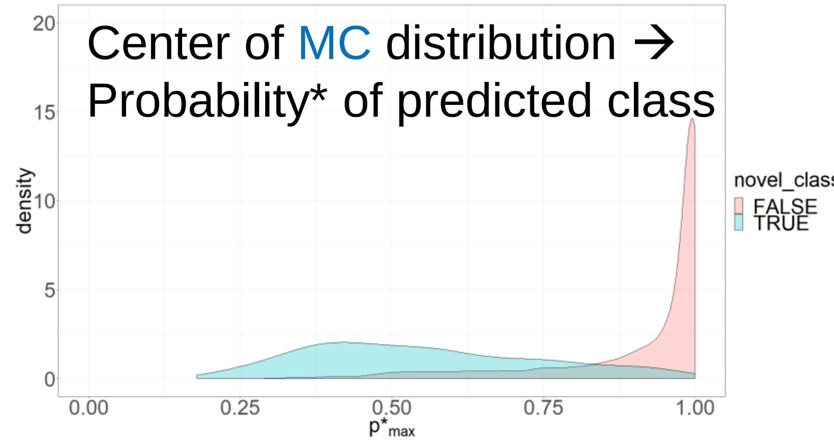
Probability distribution from MC dropout runs

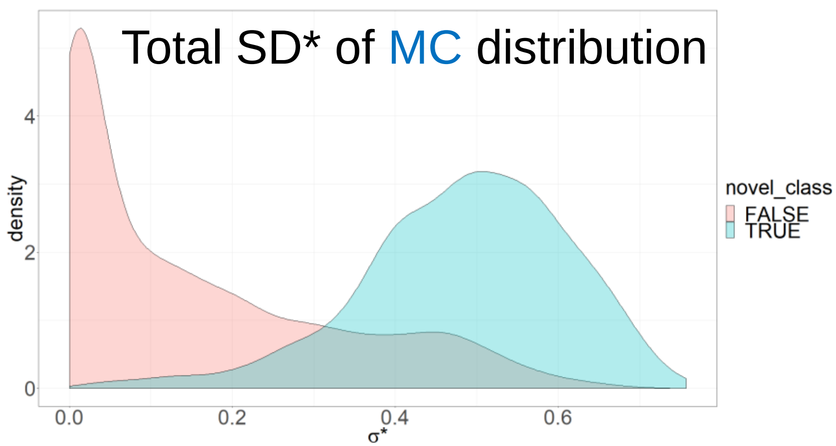
Image with known class 15

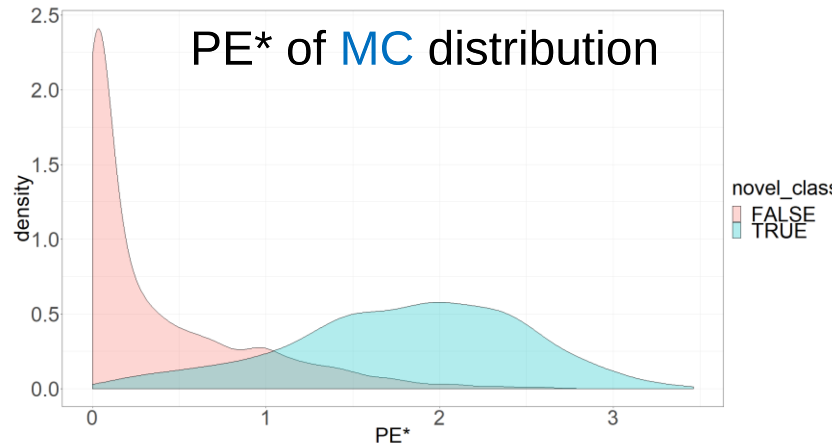


Do known/novel classes yield different values for probability and uncertainty measures?

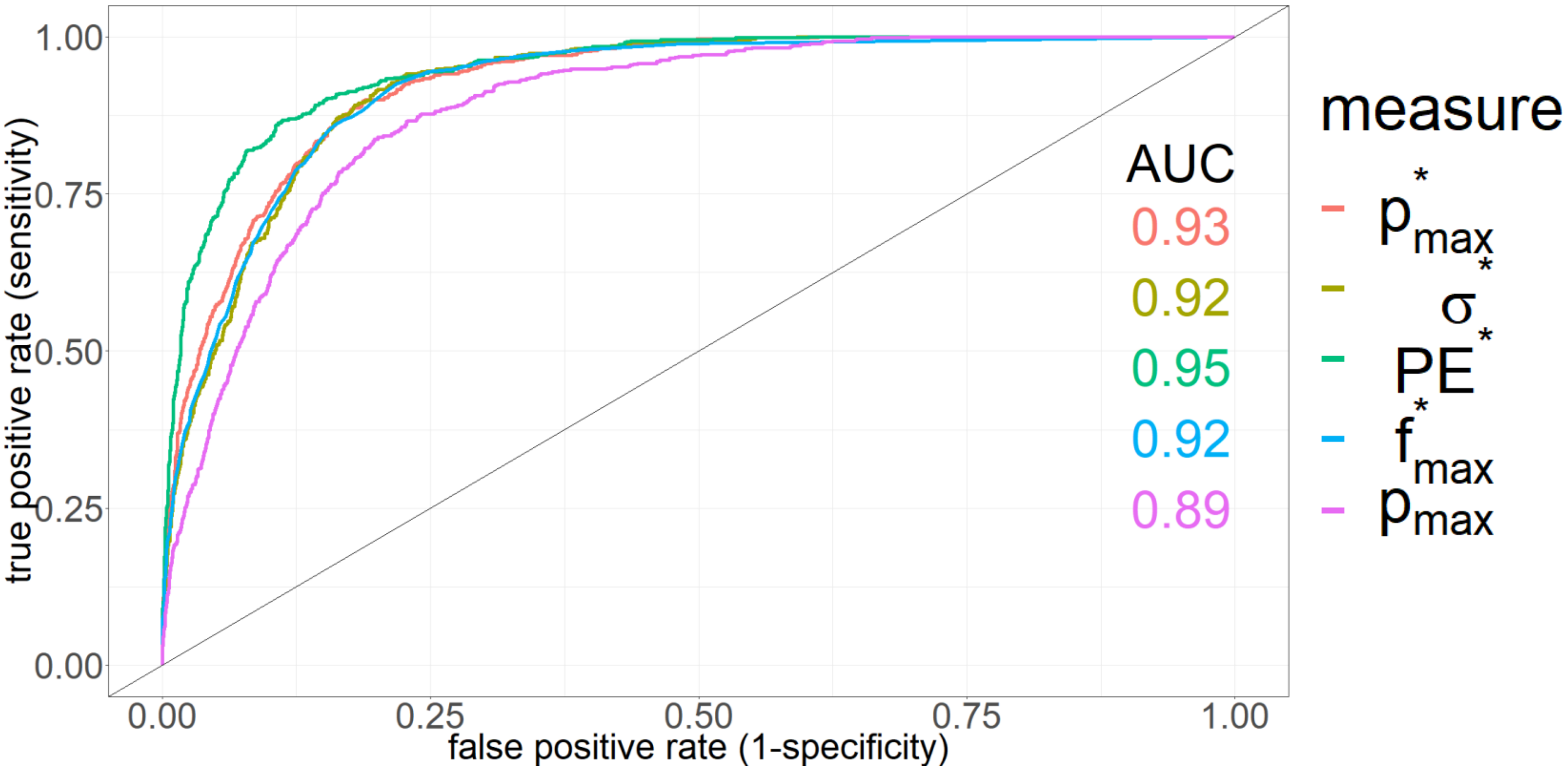








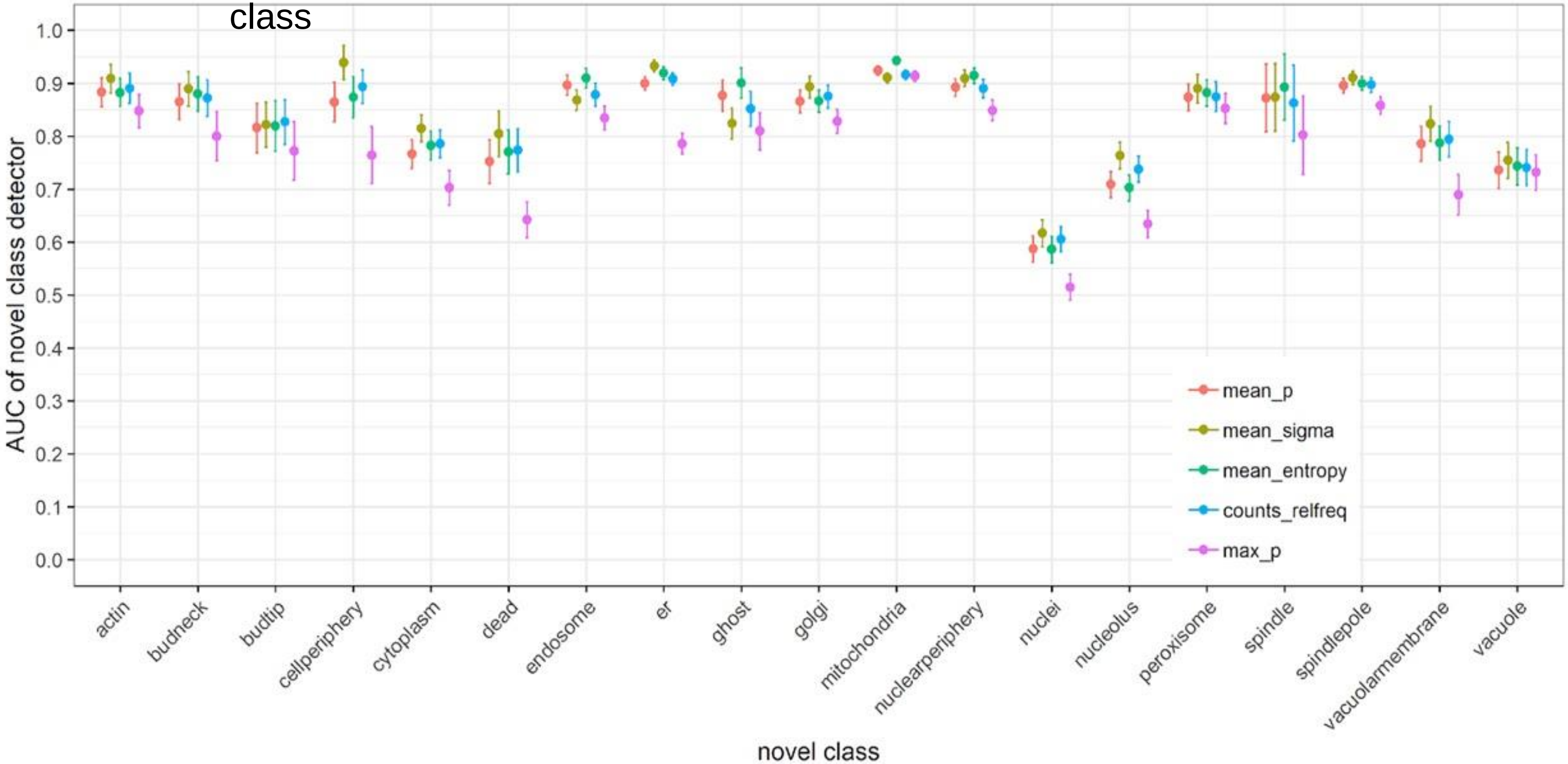
How good are novel/unusual classes identifiable?



All **MC Dropout based appoaches are superior compared** to the non-MC approach.

Dropout uncertainty measures outperform traditional CNN probability

CV experiment where each phenotyp is once in the role of the novel class



Creating custom layers in keras

```
from keras.layers.core import Lambda # needed to build the custom layer
from keras import backend as K #Now we have access to the backend (could be tensorflow,
```

Define your custom function

```
def mcdropout(x):
  #return tf.nn.dropout(x=x, keep_prob=0.33333) #using TensorFlow
  return K.dropout(x, level=0.5) # beeing agnostic of the backend
```

Here you could do anything possible in TF

`tf.add(x, 10)`

…

Include your custom function as a layer

```
model = Sequential()
model.add(Lambda(mcdropout, input_shape=(5,)))
#model.add(Dense(10))
#... Usually you would have many more layers
model.compile(loss='categorical_crossentropy',optimizer='adam')
```

https://github.com/oduerr/dl_tutorial/blob/master/tensorflow/keras/using_tf_in_keras.ipynb

Conclusion

MC Dropout during test time

→ yields uncertainty measures for each individual classification

→ helps to identify uncertain cases

→ allows to indicate novel classes

→ yields new probability estimates leading to higher accuracy