TDT4171 Artificial Intelligence Methods

Lecture 9 – More on Deep Learning

Norwegian University of Science and Technology

Helge Langseth
Gamle Fysikk 255
helge.langseth@ntnu.no

NTNU

Outline

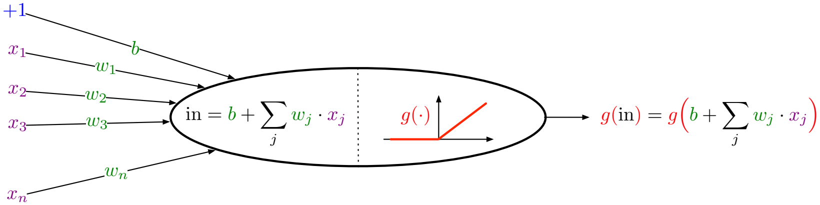
Summary from last time

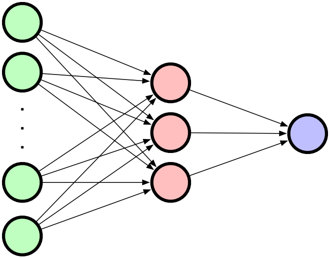
- **Learning:** Use data to generate (or improve) a representation.
- **Inductive learning hypothesis:** "Old data has relevance for new problems"
- **Neural networks:** Capable structures defined by combining many simple computational units
  - Simple perceptrons
  - Layered models $\rightarrow$ Feed forward networks
  - DL models can also contain more complicated structures
- **Learning:** Define a loss, minimize using gradient descent
- **Overfitting:** A model overfits if it does well on the training data, but fails to generalize
  - Prefer simplicity
  - Can be enforced using, e.g., norm penalties; dropout

Reference group meeting:

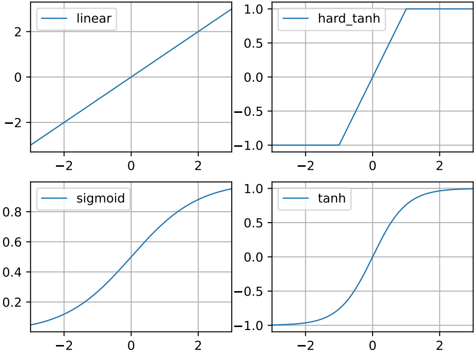
- Piazza
- Assignment deadlines
- Time-keeping

Model structure: Layers of simple nodes



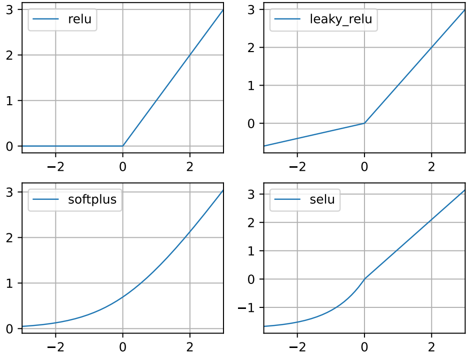


Model structure: Activations for hidden nodes



- Classic activation functions, chosen for simplicity (“`linear`”, “`hard_tanh`”) or biological plausibility (“`sigmoid`”) – the latter has a sibling that also generates negative values (“`tanh`”).
- Sigmoids no longer in much use (for hiddens) due to diminishing gradients.

Model structure: Activations for hidden nodes



- The "`relu`" (with extensions) very popular nowadays.
- Gradients zero → "`leaky_relu`"; Derivative undefined at zero → "`softplus`"; Size explosion → "`selu`".
- Finding new activation functions is still an active research field.

Model structure: Activations for output nodes

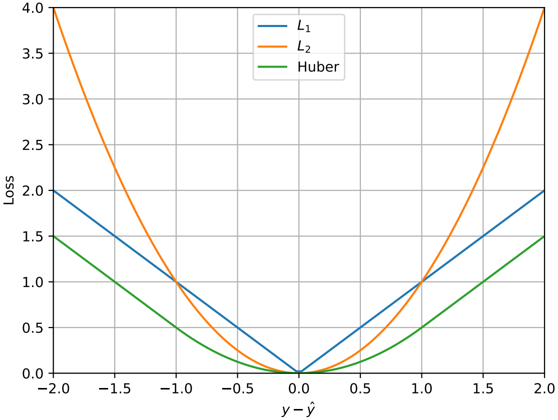
- **Regression problems:** Depending on output range, typically "`linear`" (for unbounded), "`relu`" (for positive) or "`sigmoid`"/"`tanh`" with proper scaling (for ranged)
- **Classification problems:** Nice if output layer is a probability distribution (non-negative values, sum to 1).
  - Assume we have K classes, and $\textbf{in} = (\text{in}_1, \ldots, \text{in}_K)$ be pre-activation values. Then

$$\texttt{softmax}(\textbf{in})_i = \frac{\exp(\text{in}_i)}{\sum_{j=1}^{K} \exp(\text{in}_j)}.$$

  - No constraints on $(\text{in}_1, \ldots, \text{in}_K)$ wrt. range of values: $\textbf{in} \in \mathbb{R}^K \Rightarrow \texttt{softmax}(\textbf{in})_i \geq 0, \sum_{k=1}^{K} \texttt{softmax}(\textbf{in})_k = 1.$
  - Defines a "competition" between output nodes (i.e., classes).

Model structure: Loss functions

- **Regression problems:** Typically L_p-loss: $\mathcal{L}(y, \hat{y}) = |y - \hat{y}|^p$ for $p = 1$ or 2 (or Huber-loss; a combo).



Model structure: Loss functions

- **Regression problems:** Typically L_p-loss: $\mathcal{L}(y, \hat{y}) = |y - \hat{y}|^p$ for $p = 1$ or 2 (or Huber-loss; a combo).
- **Classification problems:** Typically cross entropy loss $\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{k=1}^{K} y_k \cdot \log_2(\hat{y}_k)$. Here, $\boldsymbol{y}$ is a one-hot-encoding over the classes (a vector of length K, with all zeros except the position for the correct class, which is one).

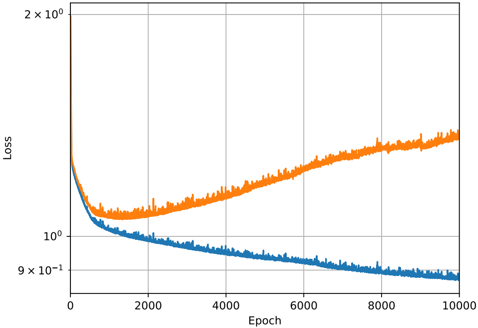
Predictions	$\hat{\mathbf{y}}$:	0.20	0.25	0.55
One-hot encoding	$\mathbf{y}$:	0	1	0
	$-y_k \log_2(\hat{y}_k)$:	0	2	0

Total loss (in bits): 2.0

Predictions	$\hat{\mathbf{y}}$:	.02	.05	.93
One-hot encoding	$\mathbf{y}$:	0	0	1
	$-y_k \log_2(\hat{y}_k)$:	0	0	0.1

Total loss (in bits): 0.1

Model structure: Regularization



- Typical example of **overfitting**: Training loss (blue) keeps going down, yet validation loss (orange) starts increasing.
- Obvious strategy: **Early stopping**.
- Additional problem: Gap between training and validation loss indicate need for **regularization**.

Model structure: Regularization (cont'd)

Definition of Regularization:

Reduction of **testing** error while maintaining a low **training** error.

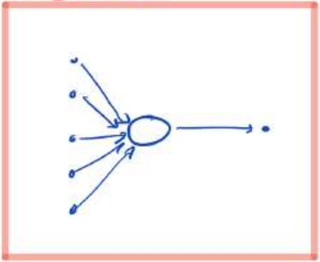
Motivation:

- Excessive training does reduce training error, but often at the expense of higher testing error.
- Flexible model classes (like neural nets) are prone to this.
- By effectively **memorizing** training-examples, we do not (necessarily) promote the ability to **generalize**

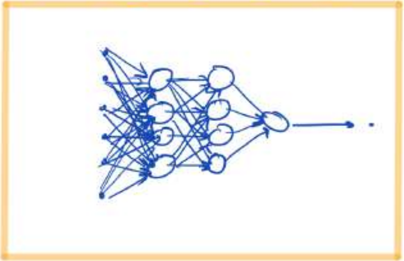
Types of regularization :

- Early stopping
- Weight-norm regularization (see slide from last week)
- Dropout (see slide from last week)
- . . . and lots more!

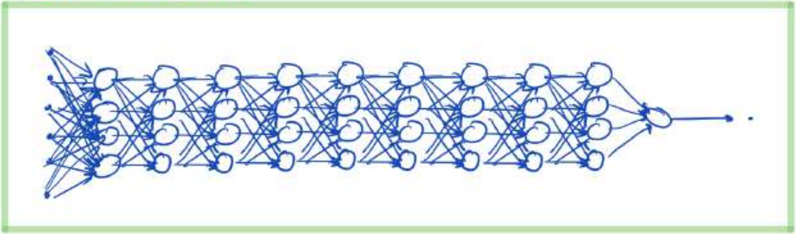
PERCEPTRON



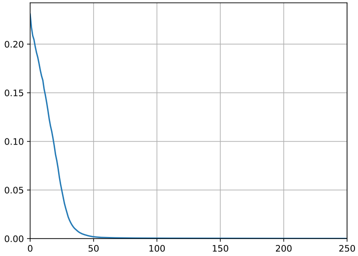
"STANDARD NEVRALT NETT"

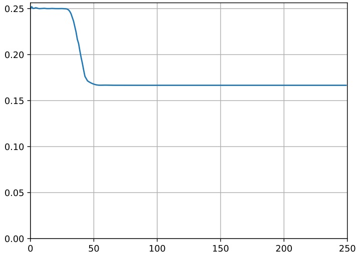


"DYP LÆRING"



Does it work?





XOR on a small network

- One epoch: All possible XOR configs.
- Model is $2 \rightarrow 2 \rightarrow 1$. Small, so hard to learn.
- All weights randomly initialized in [-1, 1].
- Sometimes learning fails, in this case initialization was crucial.

How to make it work

1. You need to be willing to spend a lot of time on hyper-parameter tuning. Tools are available. Use them!
2. Be restrictive on model size.
   Too big models take longer to learn and may overfit more.
3. Don't be too restrictive on model size.
   Too small models may be unable to represent relationship.
4. Scale all input values to the same range.
5. Be careful when initializing weights: Start small!
6. Regularize
7. Monitor *training loss* and *validation loss*.
   The former should become very small, while the latter should be similar and not increase.

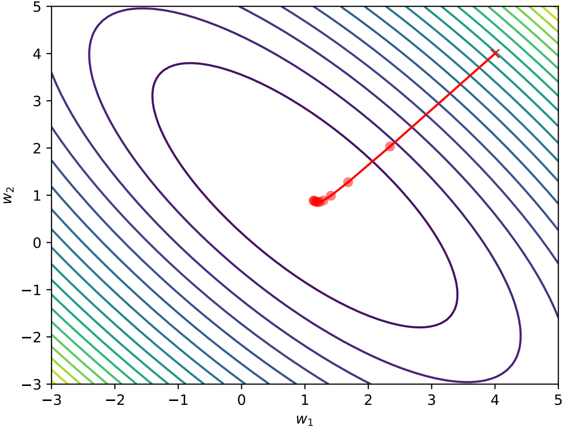
Gradient-based learning

Minimize the the objective function $\mathcal{L}$ using **partial derivatives** of $\mathcal{L}$ wrt. model weights $\boldsymbol{w}$!

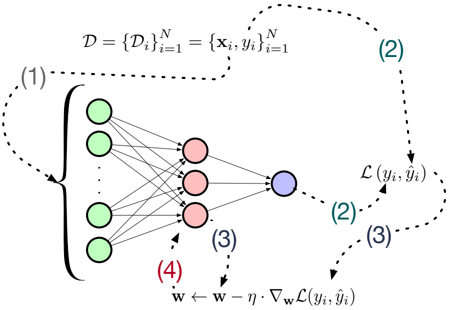
Simple setup: Learn the function $y = f(x)$:

- Data: $\mathcal{D}_i = (\boldsymbol{x}_i, y_i)$; Model weights: $\boldsymbol{w}$; Output: $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$.
- Objective: $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2 = (y - \boldsymbol{w}^\top \boldsymbol{x})^2$.
- **Note!** We will not see $y = f(\boldsymbol{x})$ for **all** possible $\boldsymbol{x}$, but will optimize based on our data $\{\mathcal{D}_i\}_{i=1}^N$. Loss is a sum over all datapoints.
- Since $\frac{\partial}{\partial w_j}\mathcal{L}(y, \boldsymbol{w}^\top \boldsymbol{x}) = -2\, x_j \cdot (y - \boldsymbol{w}^\top \boldsymbol{x})$, the move based on $(\boldsymbol{x}, y)$ would be
$$w_j \leftarrow w_j - \eta \cdot \frac{\partial}{\partial w_j}\mathcal{L}(y, \boldsymbol{w}^\top \boldsymbol{x}) = w_j - \eta \cdot 2x_j \cdot (\boldsymbol{w}^\top \boldsymbol{x} - y)$$
- Full dataset: $w_j \leftarrow w_j - \eta \cdot \sum_{i=1}^N \frac{\partial}{\partial w_j}\mathcal{L}(y_i, \boldsymbol{w}^\top \boldsymbol{x}_i)$

Gradient-based learning



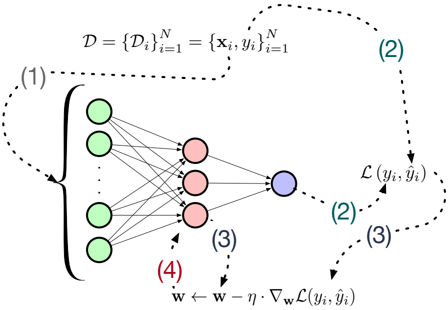
Gradient-based learning in neural nets



Steps:

1. Push $\boldsymbol{x}_i$ **forwards** to calculate $\hat{y}_i$.
2. Calculate **loss** based on $\hat{y}_i$ and observed y_i: $\mathcal{L}(y_i, \hat{y}_i)$.
3. Calculate gradients $\nabla_{\boldsymbol{w}} \mathcal{L}(y_i, \hat{y}_i)$ while moving **backwards**.
4. Update **weights**.

Gradient-based learning in neural nets



This is general purpose!

- NNs are getting increasingly complex, but the general idea is the same: Forward pass to find $\hat{y}$, backward pass to calculate $\nabla_{\boldsymbol{w}} \mathcal{L}(y_i, \hat{y}_i)$.
- Finding the gradients can be tedious and challenging, but frameworks like Tensorflow and PyTorch help us.

Gradient Descent – Details for our simple example

Recall our simple example:

Simple setup: Learn the function $f(x)$:

- Data: $\mathcal{D}_i = (\boldsymbol{x}_i, y_i)$; Model weights: $\boldsymbol{w}$; Output: $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$.
- Objective: $\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2 = (y - \boldsymbol{w}^\top \boldsymbol{x})^2$.
- This corresponds to a **perceptron** w/ **linear** activation.
- We have already seen the result:

$$w_j \leftarrow w_j - \eta \cdot \frac{\partial}{\partial w_j} \mathcal{L}(y, \boldsymbol{w}^\top \boldsymbol{x}) = w_j - \eta \cdot 2x_j \cdot (\boldsymbol{w}^\top \boldsymbol{x} - y)$$

- This is the general rule for a perceptron with identity transfer.

Gradient Descent – Details for our simple example

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \\
&= \sum_{i=1}^{N} \frac{\partial}{\partial w_j} (y_i - \hat{y}_i)^2 \\
&= \sum_{i=1}^{N} 2(y_i - \hat{y}_i) \frac{\partial}{\partial w_i} (y_i - \hat{y}_i) \\
&= 2 \sum_{i=1}^{N} (y_i - \hat{y}_i) \frac{\partial}{\partial w_j} (y_i - \boldsymbol{w}^\intercal \boldsymbol{x}_i) \\
&= -2 \sum_{i=1}^{N} (y_i - \hat{y}_i) \left(\frac{\partial}{\partial w_j} \sum_{l} w_l \cdot x_{i,l} \right) \\
\frac{\partial \mathcal{L}}{\partial w_j} &= -2 \sum_{i=1}^{N} x_{i,j} \cdot (y_i - \hat{y}_i)
\end{aligned}
$$

Gradient Descent – The algorithm for simple example

Gradient-Descent($\mathcal{D}$, η)

Each training example is a pair of the form $\langle \boldsymbol{x}, y \rangle$, where $\boldsymbol{x}$ is the vector of input values, and y is the target output value. η is the learning rate (e.g., $.05$).

1. Initialize each w_j to some small random value
2. Until the termination condition is met:
   1. Initialize: $\Delta w_j \leftarrow 0$.
   2. For each $\langle \boldsymbol{x}_i, y_i \rangle$ in $\mathcal{D}$:
      - Send $\boldsymbol{x}_i$ through the network and compute the output $\hat{y}_i$
      - For each linear unit weight w_j: $\Delta w_j \leftarrow \Delta w_j - 2(y_i - \hat{y}_i) \cdot x_{i,j}$
   3. For each linear unit weight w_j: $w_j \leftarrow w_j - \eta \cdot \Delta w_j$

Same algorithm skeleton works for other activation functions, as long as we can calculate $\frac{\partial \mathcal{L}}{\partial w_j}$.

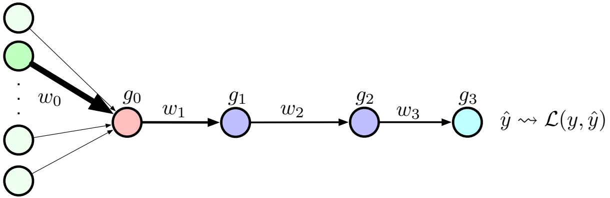
Gradient Descent: Perceptron with transfer functions

Assume the logistic transfer function:

$$g(t) = \frac{1}{1+\exp(-t)}, \quad g'(t) = g(t) \cdot \left[1 - g(t)\right].$$

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \\
&= 2 \sum_{i=1}^{N} (y_i - \hat{y}_i) \frac{\partial}{\partial w_j} \left(y_i - g\left(\boldsymbol{w}^\mathsf{T} \boldsymbol{x}_i\right)\right) \\
&= 2 \sum_{i=1}^{N} (y_i - \hat{y}_i) \cdot (-1) \cdot x_{i,j} \cdot g'\left(\boldsymbol{w}^\mathsf{T} \boldsymbol{x}_i\right) \\
\frac{\partial \mathcal{L}}{\partial w_j} &= -2 \sum_{i=1}^{N} x_{i,j} \cdot (y_i - \hat{y}_i) \cdot \underbrace{\hat{y}_i \cdot (1 - \hat{y}_i)}_{\text{This is new from the “simple example”}}
\end{aligned}$$

Backpropagation Algorithm – Single chain

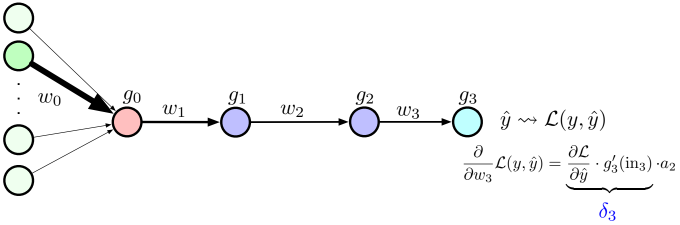


Notation for node j:

- Weight into node is w_j; its activation function $g_j(\cdot)$.
- in_j as the weighted input.
- a_j is output from node j, $a_j = g_j(\text{in}_j)$ and $\text{in}_{j+1} = w_{j+1} \cdot a_j = w_{j+1} \cdot g_j(\text{in}_j)$.

Goal: Calculate $\nabla_{\boldsymbol{w}} \mathcal{L}(y, \hat{y}) = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_3}\right]^{\mathsf{T}}$.

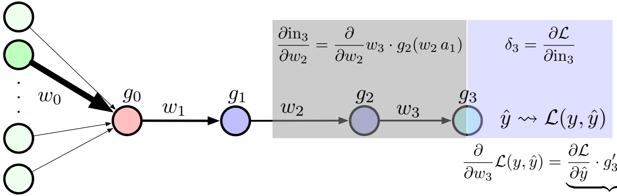
Backpropagation Algorithm – Single chain



Calculation:

- Start from back: $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\text{in}_3) \cdot a_2$; $\text{in}_3 = w_3\, a_2$.
  $\delta_3 \leftarrow \frac{\partial \mathcal{L}}{\partial \text{in}_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\text{in}_3)$. $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \text{in}_3} \frac{\partial \text{in}_3}{\partial w_3} = \delta_3\, a_2$.

Backpropagation Algorithm – Single chain



$$\frac{\partial}{\partial w_3}\mathcal{L}(y,\hat{y}) = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\text{in}_3)}_{\delta_3} \cdot a_2$$

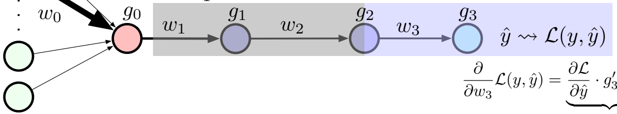
$$\frac{\partial}{\partial w_2}\mathcal{L}(y,\hat{y}) = \underbrace{w_3 \cdot \delta_3 \cdot g_2'(\text{in}_2)}_{\delta_2} \cdot a_1$$

Calculation:

- Start from back: $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\mathsf{in}_3) \cdot a_2$; $\mathsf{in}_3 = w_3\, a_2$.
  $\delta_3 \leftarrow \frac{\partial \mathcal{L}}{\partial \mathsf{in}_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\mathsf{in}_3)$. $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \mathsf{in}_3} \frac{\partial \mathsf{in}_3}{\partial w_3} = \delta_3\, a_2$.
- $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \mathsf{in}_3} \frac{\partial \mathsf{in}_3}{\partial w_2} = \delta_3 \cdot w_3 \frac{\partial}{\partial w_2} \cdot g_2(w_2 \cdot a_1) = \delta_3 \cdot w_3 \cdot g_2'(\mathsf{in}_2) \cdot a_1$;
  $\delta_2 \leftarrow \frac{\partial \mathcal{L}}{\partial \mathsf{in}_2} = \delta_3 \cdot w_3 \cdot g_2'(\mathsf{in}_2)$.

Backpropagation Algorithm – Single chain

$$\frac{\partial}{\partial w_1}\mathcal{L}(y,\hat{y}) = \overbrace{w_2 \cdot \delta_2 \cdot g_1'(\mathrm{in}_1)}^{\delta_1} \cdot a_0$$



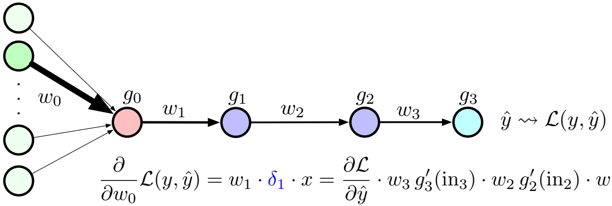
$$\frac{\partial}{\partial w_3}\mathcal{L}(y,\hat{y}) = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\mathrm{in}_3)}_{\delta_3} \cdot a_2$$

$$\frac{\partial}{\partial w_2}\mathcal{L}(y,\hat{y}) = \underbrace{w_3 \cdot \delta_3 \cdot g_2'(\mathrm{in}_2)}_{\delta_2} \cdot a_1$$

Calculation:

- Start from back: $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\mathrm{in}_3) \cdot a_2$; $\mathrm{in}_3 = w_3\, a_2$.
  $\delta_3 \leftarrow \frac{\partial \mathcal{L}}{\partial \mathrm{in}_3} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot g_3'(\mathrm{in}_3)$. $\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial \mathrm{in}_3} \frac{\partial \mathrm{in}_3}{\partial w_3} = \delta_3\, a_2$.
- $\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \mathrm{in}_3} \frac{\partial \mathrm{in}_3}{\partial w_2} = \delta_3 \cdot w_3 \frac{\partial}{\partial w_2} \cdot g_2(w_2 \cdot a_1) = \delta_3 \cdot w_3 \cdot g_2'(\mathrm{in}_2) \cdot a_1$;
  $\delta_2 \leftarrow \frac{\partial \mathcal{L}}{\partial \mathrm{in}_2} = \delta_3 \cdot w_3 \cdot g_2'(\mathrm{in}_2)$.
- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \mathrm{in}_2} \frac{\partial \mathrm{in}_2}{\partial w_1} = \delta_2 \cdot w_2 \cdot g_1'(\mathrm{in}_1) \cdot a_0$.

Backpropagation Algorithm – Single chain

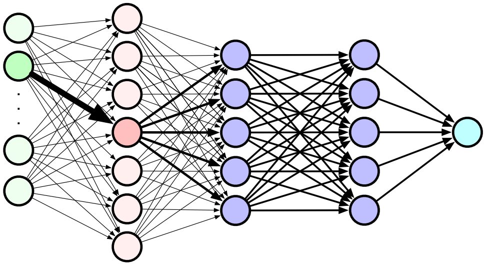


$$\frac{\partial}{\partial w_0}\mathcal{L}(y,\hat{y}) = w_1 \cdot \delta_1 \cdot x = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot w_3\, g_3'(\mathrm{in}_3) \cdot w_2\, g_2'(\mathrm{in}_2) \cdot w_1\, g_1'(\mathrm{in}_1) \cdot x$$

Notice:

- **Forward** for activations: $a_{j+1} = g_{j+1}(w_{j+1}\, a_j)$.
- **Backward** for derivatives:
  - $\delta_j = \frac{\partial \mathcal{L}}{\partial \mathsf{in}_j} = g_j'(\mathsf{in}_j)\, w_{j+1} \cdot \delta_{j+1}$.
  - $\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial \mathcal{L}}{\partial \mathsf{in}_j} \frac{\partial \mathsf{in}_j}{\partial w_j} = \delta_j \cdot a_{j-1}$

Backpropagation Algorithm – General model



Layered model:

- The effect of changing w_0 will spread through many additive "paths"; all must be considered when adapting the weight.
- **The derivatives of a sum equals the sum of derivatives**, so this is actually "easy" to generalize.
- Now $\delta_j = \frac{\partial \mathcal{L}}{\mathsf{in}_j}$ is a **vector** with one element per node in layer j:
  $\delta_{j,k} = g'_j(\mathsf{in}_{j,k}) \sum_\ell w_{j+1,\ell} \cdot \delta_{j+1,\ell}$.

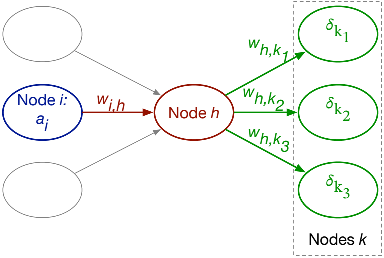
Backpropagation Algorithm – Sigmoid units

1. Initialize all weights to small random numbers.
2. Until satisfied:
   - For each training example
     1. Input the training example to the network and compute the network outputs and node activations along the way
     2. For each output unit k: $\delta_k \leftarrow \underbrace{-2(y_k - \hat{y}_k)}_{\text{Loss contrib.}} \times \underbrace{\hat{y}_k(1 - \hat{y}_k)}_{g'(\text{in}_k)}$
     3. For each hidden unit h:

$$\delta_h \leftarrow \underbrace{a_h(1 - a_h)}_{g'(\text{in}_h)} \times \underbrace{\sum_{k \in \text{outputs}} w_{h,k} \cdot \delta_k}_{\text{Unit } h\text{'s contribution to next layer's error}}$$

     4. Update each network weight $w_{i,j}$: $w_{i,j} \leftarrow w_{i,j} - \eta\, \Delta w_{i,j}$, where $\Delta w_{i,j} = \delta_j \cdot a_i$ and a_i is the activation of the node pushing information into $w_{i,j}$.

Backprop in yet another picture

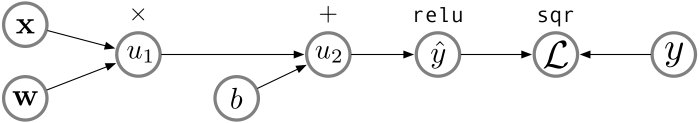


$$\delta_h \leftarrow a_h(1-a_h) \times \sum_k w_{h,k}\,\delta_k \quad \text{and} \quad \Delta w_{i,h} \leftarrow \delta_h \cdot a_i$$

Note the "locality" of the calculations:

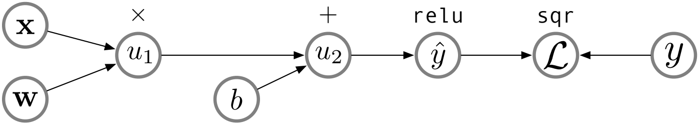
- $\Delta w_{i,h}$ only cares about Node h, its parent (Node i), and its children (Nodes k).
- The calculation of $\delta_h = \frac{\partial \mathcal{L}}{\partial \text{in}_h}$ does not care about the parent (Node i), so we can use the same value for all parents!

Backprop in DL: The computational graph



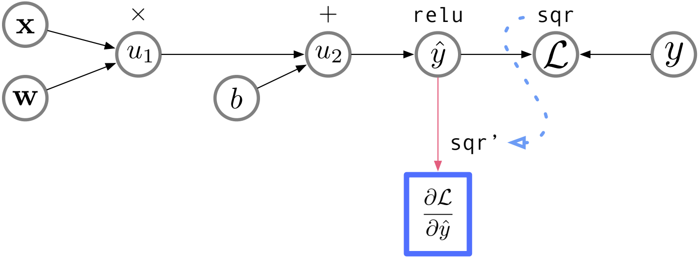
- Representation to describe computations in a directed graph. This one shows $\hat{y} = \text{relu}(\boldsymbol{x}^\intercal \boldsymbol{w} + b)$; $\mathcal{L} = (y - \hat{y})^2$.
- The layered NN architecture maps directly to the comp.graph:
  - All calculation results are nodes
  - All terms used in a node's calculations are parents.
- Gradients are found using the graph and the **chain rule**.

Backprop in DL: The computational graph

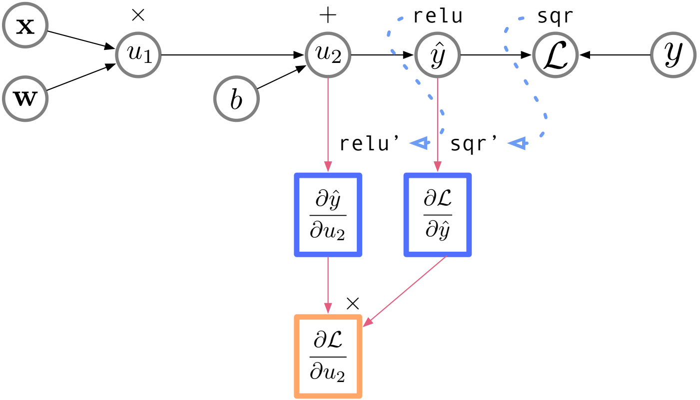


- Calculating derivatives by going backwards in the graph:
  - Do a local compute at node
  - Multiply with result from child.
  - If several children: Just sum
- Starting at $\mathcal{L}$ and ending at b gives us $\frac{\partial \mathcal{L}}{\partial b}$; starting at $\mathcal{L}$ and ending at $\boldsymbol{w}$ produces $\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$.

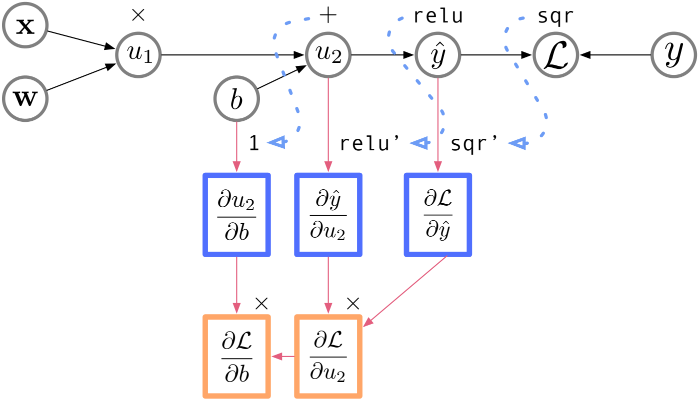
Backprop in DL: The computational graph



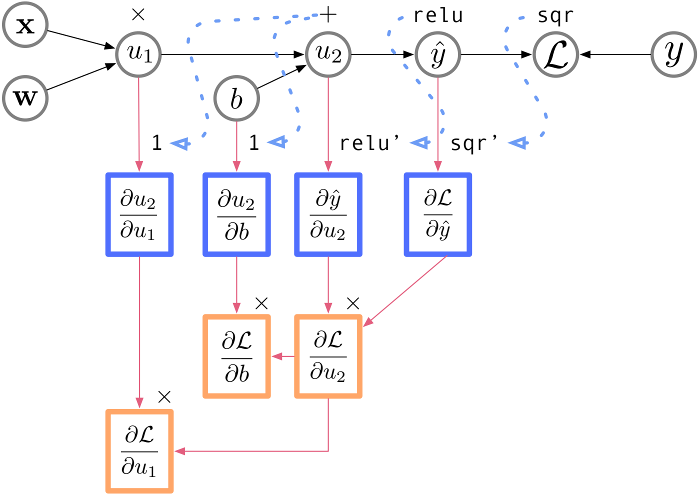
Backprop in DL: The computational graph



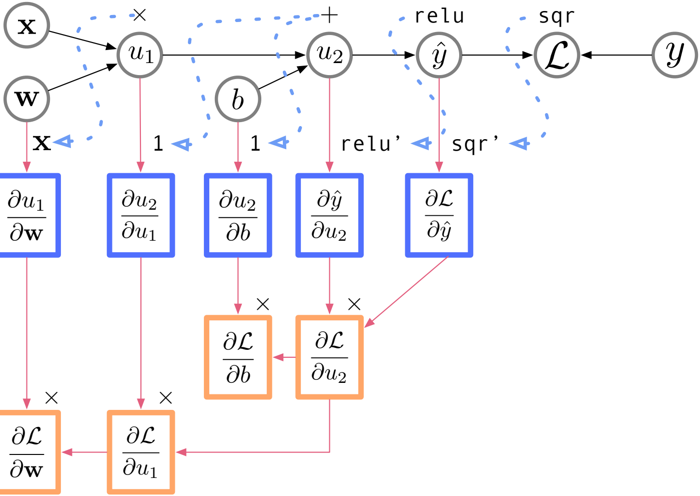
Backprop in DL: The computational graph



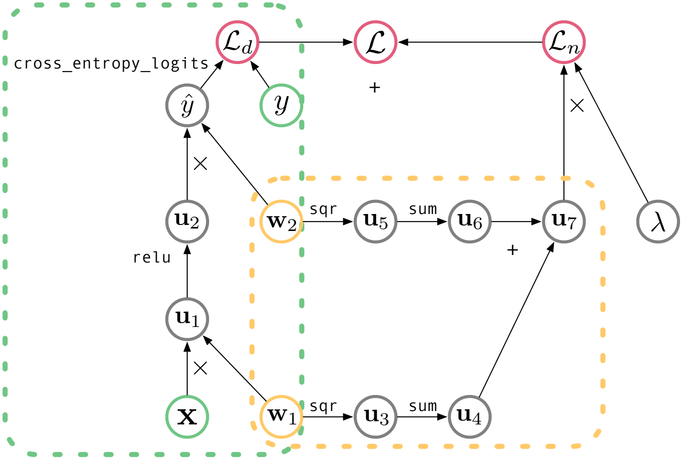
Backprop in DL: The computational graph



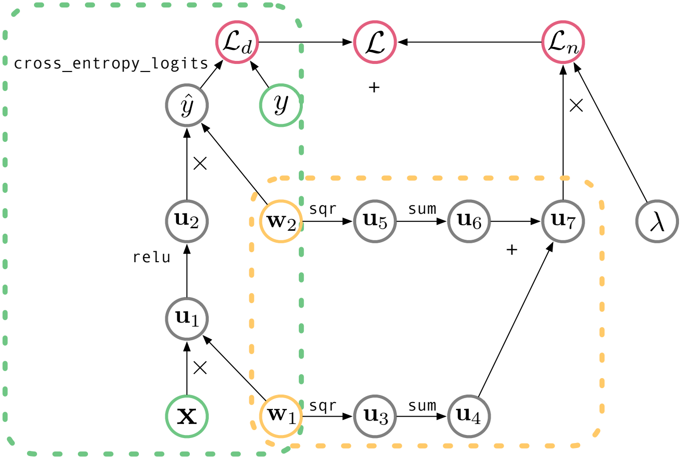
Backprop in DL: The computational graph



Computational graph for classification



Computational graph for classification



$$\mathcal{L} = \text{cross\_entropy\_logits}(y, \boldsymbol{w}_2^\intercal \overbrace{\text{relu}(\underbrace{\boldsymbol{w}_1^\intercal \boldsymbol{x}}_{\boldsymbol{u}_1})}^{\boldsymbol{u}_2}) + \overbrace{\lambda \cdot \underbrace{(\|\boldsymbol{w}_1\|_2 + \|\boldsymbol{w}_2\|_2)}_{\boldsymbol{u}_7}}^{\mathcal{L}_n}$$

Convergence of Backpropagation

Gradient descent to some local minimum:

- Perhaps not the *global* minimum ...
  - Include weight *momentum* $\alpha > 0$:

$$\Delta w_{i,j}(n) = \left(\underbrace{2\delta_j \cdot a_i}_{\text{The ``standard''}} \quad + \quad \underbrace{\alpha \cdot \Delta w_{i,j}(n-1)}_{\text{Scaled last move}} \right)$$

  - Train multiple nets with different initial weights
  - Batching/stochastic gradient descent

Nature of convergence – depending on g (here: logistic):

- Initialize weights near zero
  $\rightarrow$ Therefore, initial networks *near-linear*.
- Increasingly non-linear functions possible as training progresses.

Backpropagation summary

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well
- Minimizes error over *training* examples
  - Will it generalize well to subsequent examples? **Overfitting...**
- Training can take thousands of iterations $\rightarrow$ **slow!**
- Using network after training is very fast

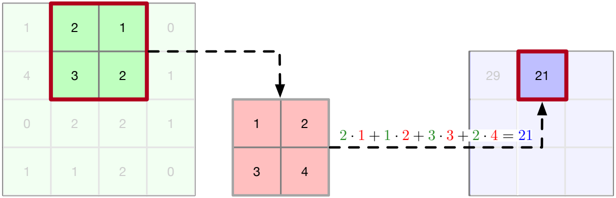
Convolutional Neural nets

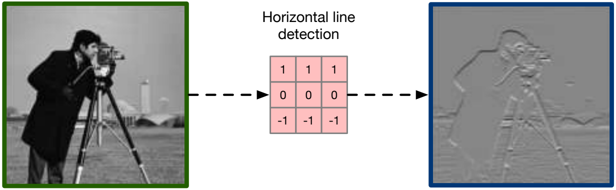
- **Goal:** Scale up to process very large images/videos
  - Sparse connections
  - Parameter sharing
  - Automatically generalize across spatial translations of inputs
  - Applicable to any input laid out on a grid (1-D, 2-D, 3-D, . . .) and other data with spatial structure

- **Key idea:** Replace/replicate flattened representations and matrix multiplication with convolution that respect locality of information. **Everything else stays the same:**
  - Optimization criteria
  - Training algorithm
  - And so on . . .

Convolutions

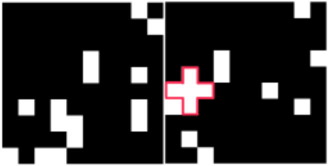
Std. definition: $(f_1 * f_2)(t) = \int_{-\infty}^{\infty} f_1(\tau) \cdot f_2(t-\tau) \mathrm{d}\tau$



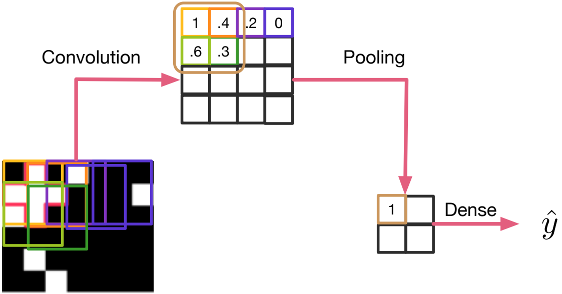


Learning convolutions

Data: Random binary images, some with a cross

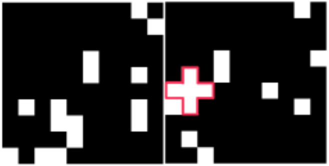


Model:



Learning convolutions

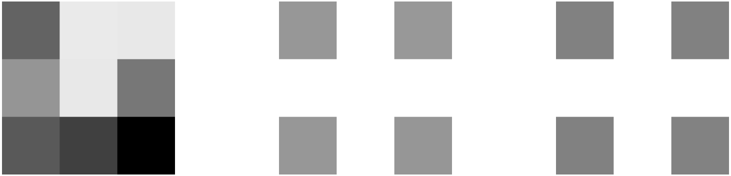
Data: Random binary images, some with a cross



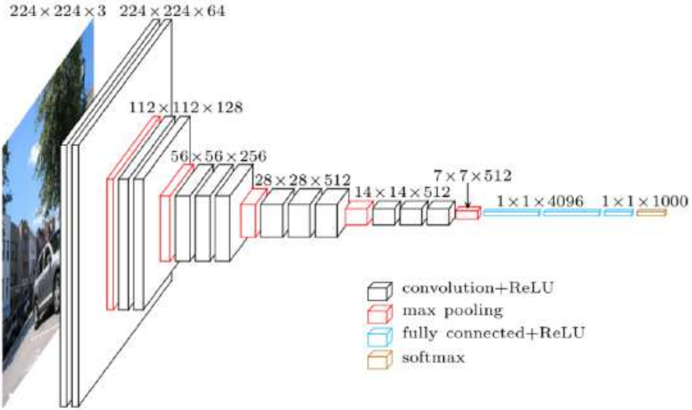
Model:

Single 3×3 convolutional kernel, one filter, with `relu`,
max-pooling down to 2×2, then flatten and logistic on output.

Results: Kernel after 1, 10, and 20 epochs:



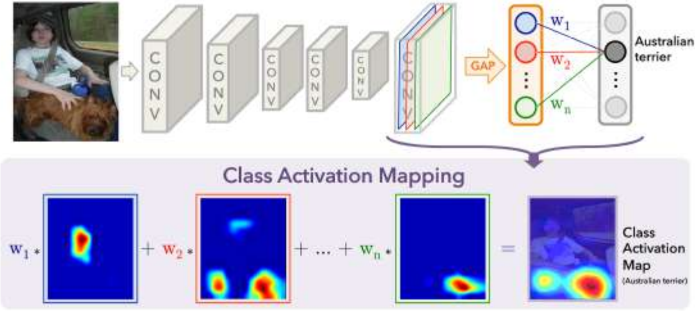
Classic CNN model architecture: VGG16 from 2014



XAI: Understanding what the model does

Early example: Class Activation Mapping (CAM, 2015)

- After last convolution: Average over spatial positions, so each filter is represented by one variable
- Learn classifier over that $\rightarrow$ weights per filter and class.
- Use location-based firing combined with filter weights to find class activation map.



Recurrent Neural Networks (RNNs)

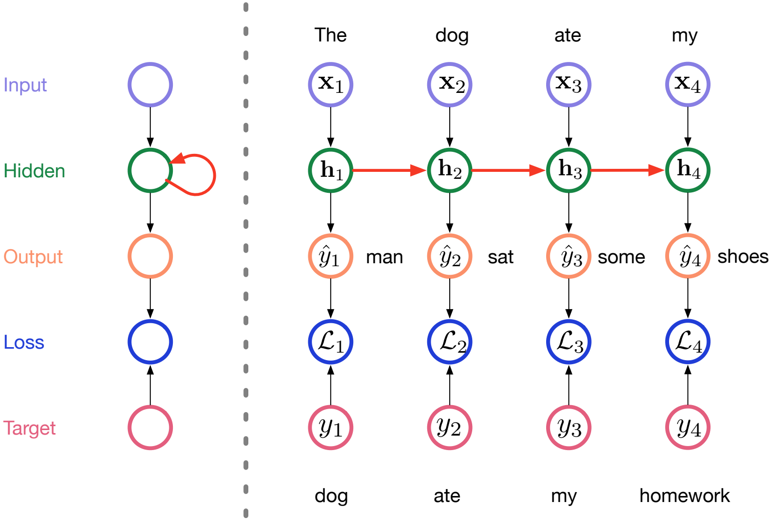
Typical use-cases for RNNs

- Inputs can be variable-length sequences; order is important; long-term dependencies.; parameter sharing/stationarity.
- Analysis of time-series data (e.g., measurements stock tickers over time) and predictions over those
- Sequential data, in particular anything with language: Modeling, Generation, Translation, Recognition, . . .
- **Transformers** (attention-based models) and **conv-nets** are getting increasingly popular in this domain.

RNNs

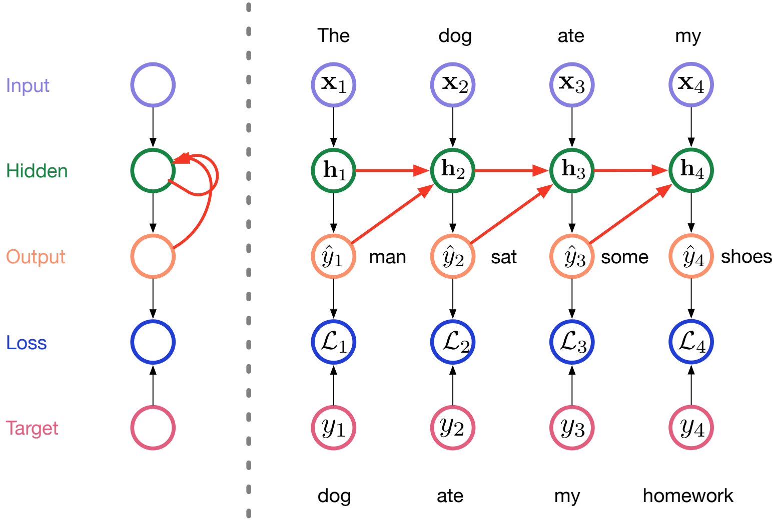
- A neural network that models sequential data (as HMMs)
- Many versions exist; sometimes tailor-made to accommodate particular signal flows (resembling DBN modelling task).

Standard architectures: Here for next-word prediction



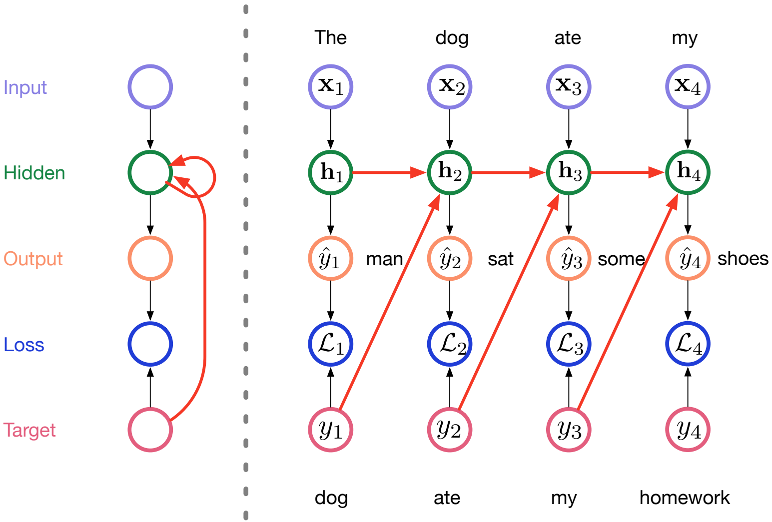
Plan: HMM-like structure

Standard architectures: Here for next-word prediction



Plan: Push generated tokens

Standard architectures: Here for next-word prediction



Plan: Push *correct* tokens during training; generated during test.

Standard architectures: Here for next-word prediction

- Other architectures (beyond "one-to-one-repeated") exist, like
  - **Many-to-one:** Sequence classification
  - **One-to-many:** Captioning
  - **Sequence-to-sequence:** Translation
- **Problematic issue:** Track long-term information?

My dog disappeared. I went by my mom's house searching, had a coffee, and met my sister. She has moved away, and I don't see her that much. Then I went home, talked to my wife, and watched a ballgame. My team won. I had a good beer, one of my favourites, that is now available at the local grocery store. Then I found him! **Question:** Who was found?

  - Size of $\boldsymbol{h}_t$ may have to extremely large to transport all that is needed to know. Hard to learn!
  - Can we somehow **enforce structure** on the model to simplify information aggregation?
    - Yes, using LSTMs! (Next slide)
    - Yes, using transformers! (in two weeks)

Long Short Term Memory/LSTM

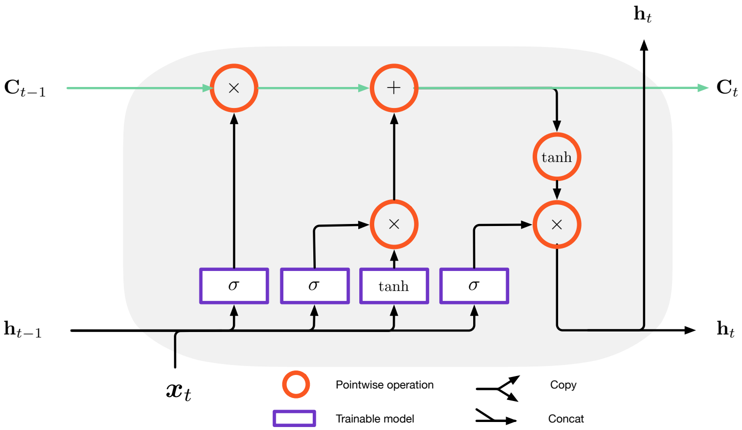
LSTM background:

- Invented by Hochreiter and Schmidhuber in 1997.
- Lots of different versions; ; massively popular since 2009.
- Can be combined with other layers (e.g., convolution)

Modelling components:

- Cell-state $\boldsymbol{C}_t$ and hidden state $\boldsymbol{h}_t$. Both passed through time.
- $\boldsymbol{C}_t \leftarrow \alpha_C([\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]) \cdot \boldsymbol{C}_{t-1} + \beta_C([\boldsymbol{h}_{t-1}, \boldsymbol{x}_t])$: Modulate based on input and hidden, add based on input and hidden;
- $\boldsymbol{h}_t \leftarrow \alpha_h([\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]) \cdot \tanh(\boldsymbol{C}_t)$: Modulate transformed $\boldsymbol{C}_t$ using input and hidden; $\boldsymbol{h}_t$ is also the output at time t.

LSTMs (Hochreiter&Schmidhuber, 1997)



LSTM gradients need not be diminishing

$\rightarrow$ Can learn long-term relationships.

Summary

- **Deep learning** is the most prominent ML technique nowadays.
- **Main reasons for success:** Data, compute, algorithmic developments.
- Model tuning using **gradient-based** techniques. Backprop is the general-purpose workhorse.
- **Modelling** is still important: CNNs, RNNs incl. LSTMs, (and later: Transformers) are defined to incorporate specific biases.