COMP 432 Machine Learning

Neural Networks (Part 3)

Computer Science & Software Engineering
Concordia University, Fall 2024

Summary of the last episode….

What we have seen **last time**:

- Basic Concepts for Neural Networks (vanishing gradient, initialization, normalization)
- Convolutional Neural Networks

What we are going to learn **today**:

- Regularization
- Recurrent Neural Networks

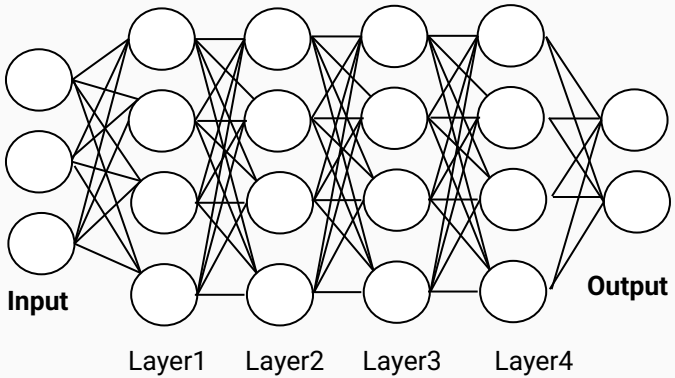
Regularization

Regularization

- In machine learning we want our model to perform well on **new inputs** (not observed during training).
- This ability is called **generalization**.
- In particular, we want to avoid **overfitting**, which happens when the **training error** is **low** but the **test error** is **high**.
- Deep Neural Networks are models with **large capacity** and thus more prone to **overfitting** problems.
- **Regularization** thus plays an important role.
- In the following, we will see popular regularization techniques used in deep learning.

L2 and L1 Regularization

- **L2** and **L1 regularizations** are used in the context of deep neural networks as well.
- As mentioned in Lecture 3, they discourage complex solutions by penalizing the norm of the **weights**:



L2 Regularization:

$$\tilde{J}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(L)}) = J(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(L)}) + \lambda \sum_{l=1}^{L} ||\mathbf{W}^{(l)}||_2^2$$

$$||\mathbf{W}^{(l)}||_2^2 = \sum_{i=1}^{N_{in}} \sum_{j=1}^{N_{out}} w_{ij}^2$$

L1 Regularization:

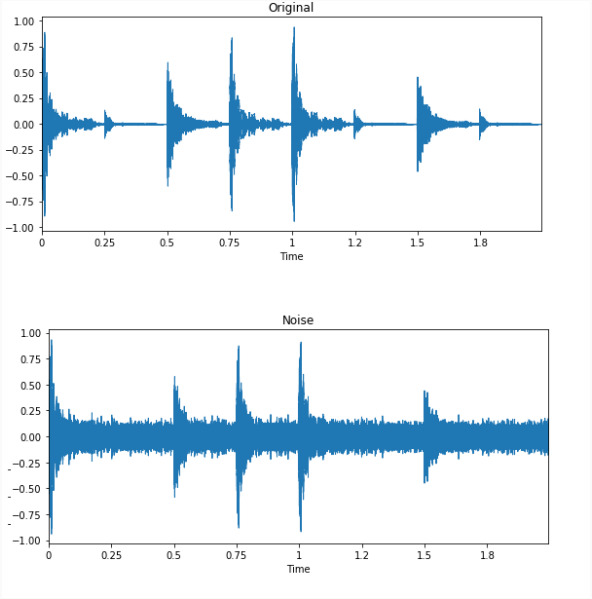
$$\tilde{J}(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(L)}) = J(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, ..., \mathbf{W}^{(L)}) + \lambda \sum_{l=1}^{L} ||\mathbf{W}^{(l)}||_1$$

$$||\mathbf{W}^{(l)}||_1 = \sum_{i=1}^{N_{in}} \sum_{j=1}^{N_{out}} |w_{ij}|$$

We apply the penalization factor to **all the weights** of the deep neural network. In principle, we can use a different regularization factor for each weight matrix. This is normally not done because it introduces too many hyperparameters.

Data Augmentation

- The best way to make a machine learning model generalize better is to train it with **more data**.
- If not possible, we can create "*fake*" training data by applying some **transformations** to the **original data**.
- This approach is called **data augmentation**. It is extremely successful for **image** and **audio**.

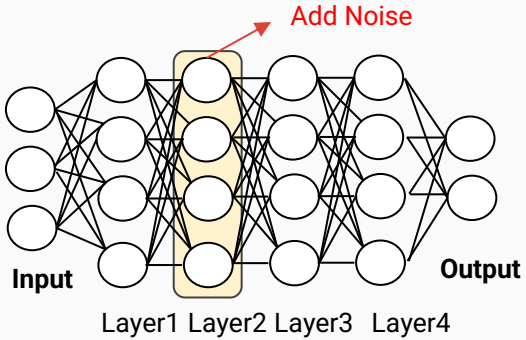


Data Augmentation

- Data augmentation can be used mainly for **classification problems.**

One must be careful not to apply transformations that change the correct class.

For instance, in digit recognition, we cannot perform rotations that turn a “6” into a “9” and vice-versa.



- Noise injection might also work when **noise** is applied to the **hidden units**.
- This can be seen as an augmentation applied at different levels of abstraction.

Label Smoothing

- In some cases, we regularize a model by making it **less overconfident** about its predictions.
- One way is to replace **hard labels** with **soft labels**.
- When computing categorical cross-entropy, we can use these soft labels instead of the hard ones:

$$NLL = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} \ln P(y_i = k \mid \mathbf{x}_i, \mathbf{w})$$

One-hot encoding (hard labels)

$$\mathbf{Y} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

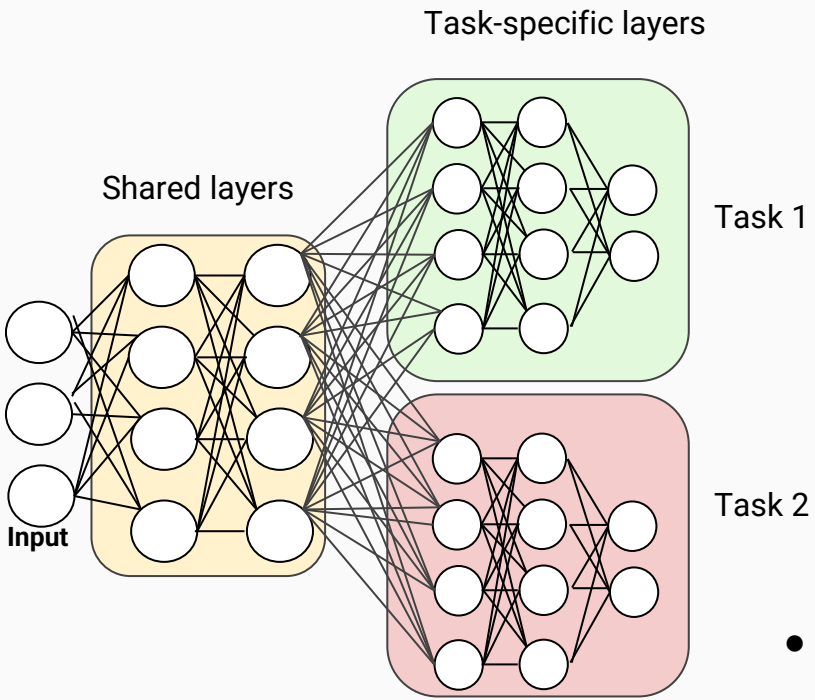
Smooth

Soft labels

$$\mathbf{Y} = \begin{bmatrix} 0.05 & 0.05 & 0.85 & 0.05 \\ 0.05 & 0.85 & 0.05 & 0.05 \\ 0.05 & 0.05 & 0.05 & 0.05 \\ 0.85 & 0.05 & 0.05 & 0.05 \end{bmatrix}$$

Multitask Learning

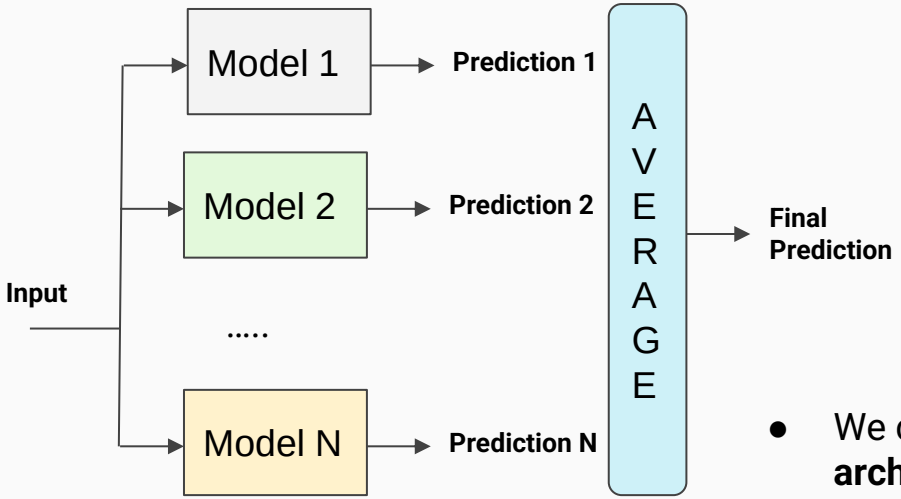
- We can improve generalization by solving **multiple related tasks** at the same time.



- This approach is called **multi-task learning**.
- One common way is to divide the architecture into a **shared** one (early layers) and a **task-specific** one (last layers).
  - Example: given a speech recording, task 1 detects phonemes and task 2 classifies speaker identities.
  - The input must be the same.
- The shared representation will be much more **robust** and **general** as trained with more data on different tasks.

Bagging

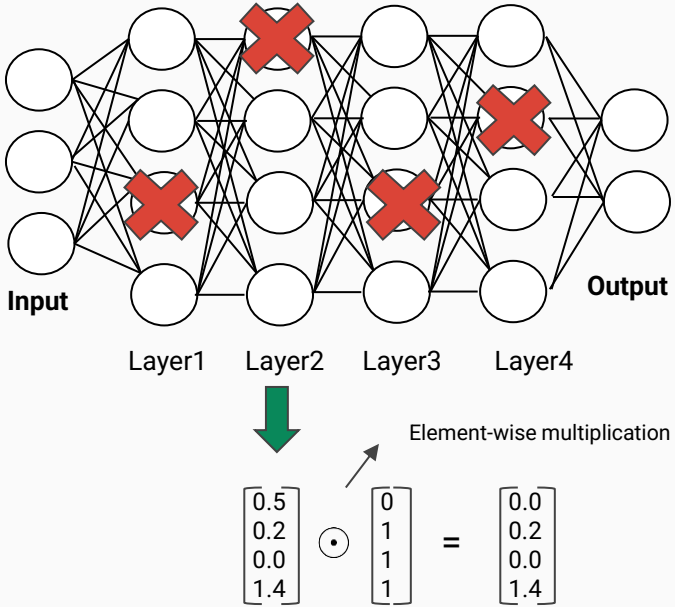
- **Bagging** is a technique that improves generalization by **combining several models**.



- We can **train** the models **independently** and **average** their predictions at **test** time.
- If the models have **similar performance** and provide **uncorrelated errors**, the final performance is better than the single predictions.
- We can train the models with **different data** or **different architectures**.
- The main issue is that training multiple models can be very **expensive**.

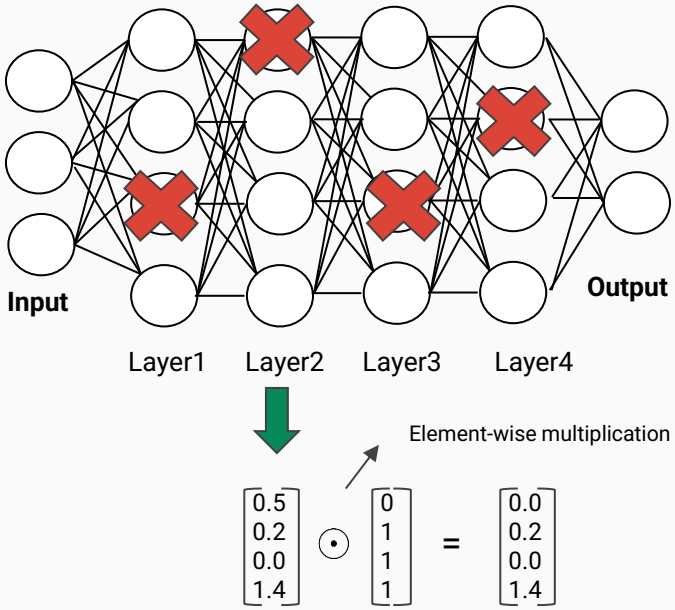
Dropout

- **Dropout** is a **regularization** method that efficiently approximates training a large number of neural networks.
- During training, some neurons are randomly ignored or "**dropped out**."



- These neurons are **temporarily removed** from the network along with their input and output connections.
- One way to implement it is to sample a **binary mask** (that contains 0s and 1s) for each layer and multiply it with the output of the neurons.
- When we multiply by **0** the neuron is **deactivated**, while when we multiply by **1** is **active**.
- The probability of dropping a neuron is called dropout rate ρ.
- For every **input** sample, we sample a **different mask**.

Dropout



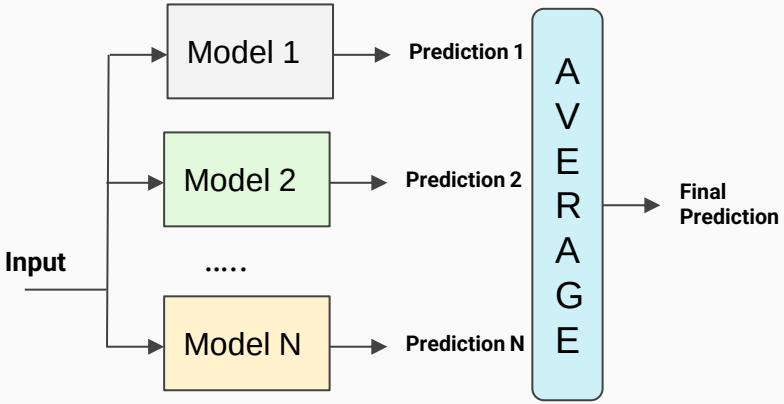
- The neural networks will be composed of **different active neurons** for every minibatch.

- We sample a “different” network for every input

- Dropout can be viewed as a cheap way to train **ensemble models** with an **exponential number** of architectures.

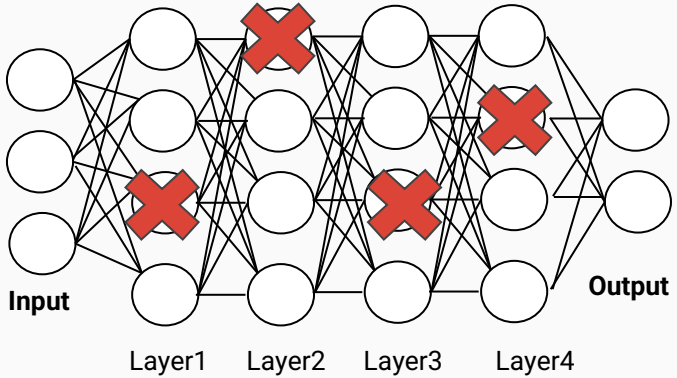
Dropout vs Bagging

Bagging



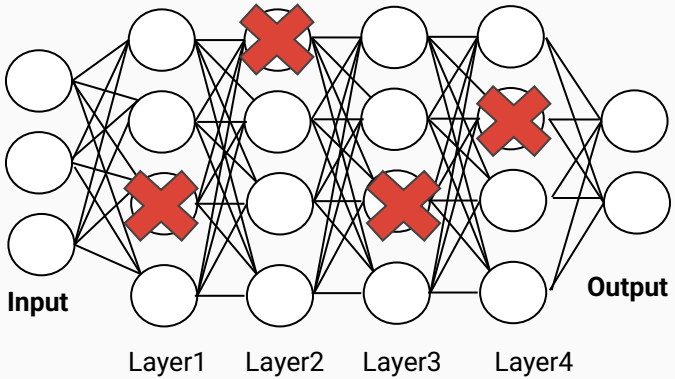
- The models **do not share** the **weights**.
- We only average a **few models.**
- We **average** the final predictions.

Dropout



- The models **share** the **weights**.
- We average an **exponential** number of models.
- We **average** internal features.

Dropout



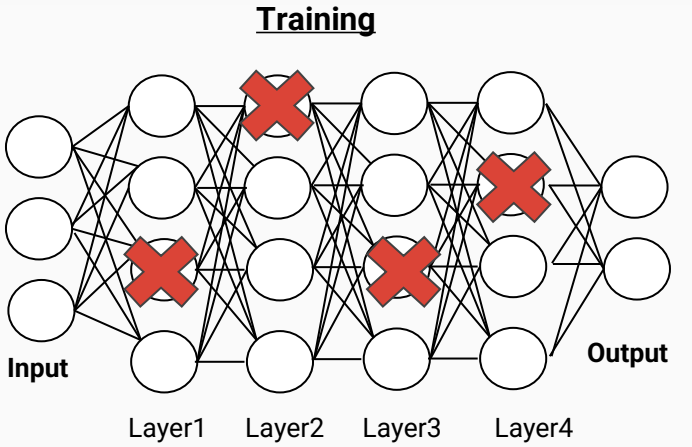
- We can also see dropout as a way to **inject noise** into the **learning** process.
- We have seen that injecting **little noise** into the learning process is **effective** in many cases (think about stochastic gradient descent or data augmentation).

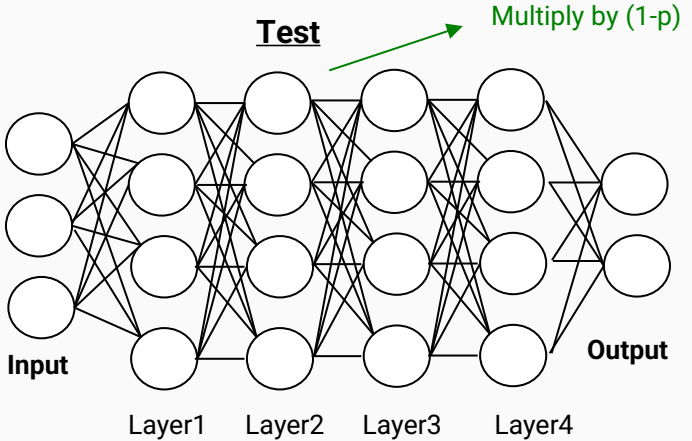
- Another “angle” to see dropout is to see it as a way to prevent **co-adaptation**:

> … units may change in a way that they fix up the mistakes of the other units. This may lead to complex **co-adaptations**. This in turn leads to **overfitting** because these co-adaptations do not generalize to unseen data. […]

Dropout prevents the neurons to rely “too much” on the output of other neurons.

Dropout

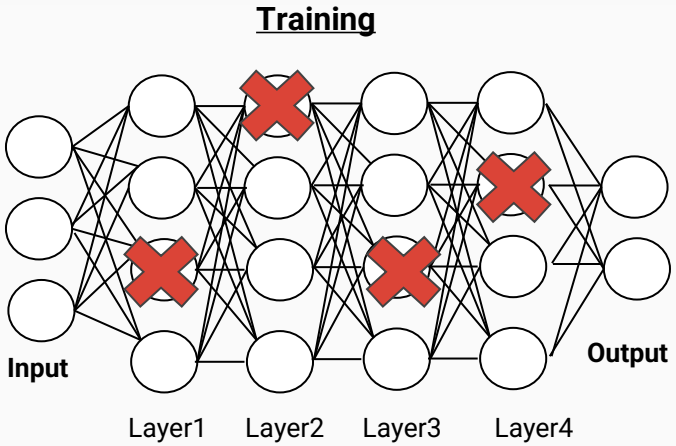


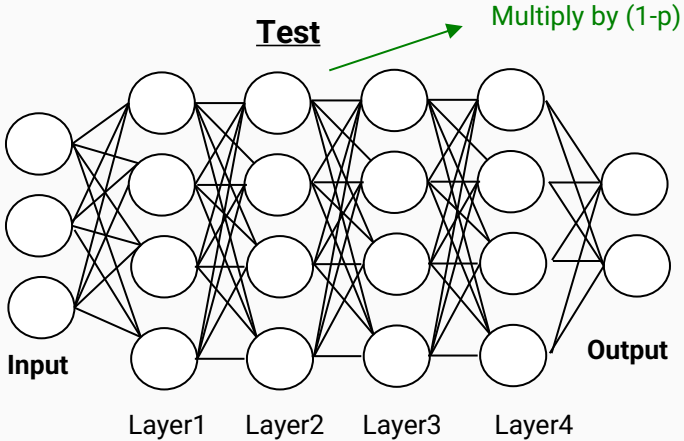


- *What do we do at test/inference time?*
- Dropout is **not used at test time.**

- At test time, we want to use **all the ensembled models** and not just one.
- At test time, **all** the **neurons** are thus **active**, and we have to **scale** their outputs properly.
- If a neuron is dropped with probability **p** during training, the outgoing weights of that neuron are multiplied by **1-p** at **test time**.

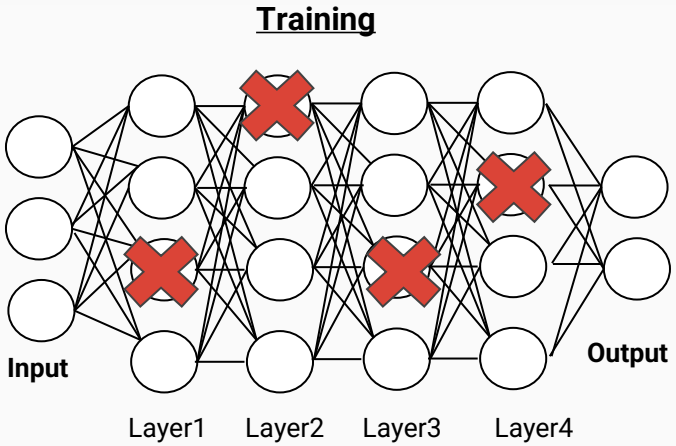
Dropout

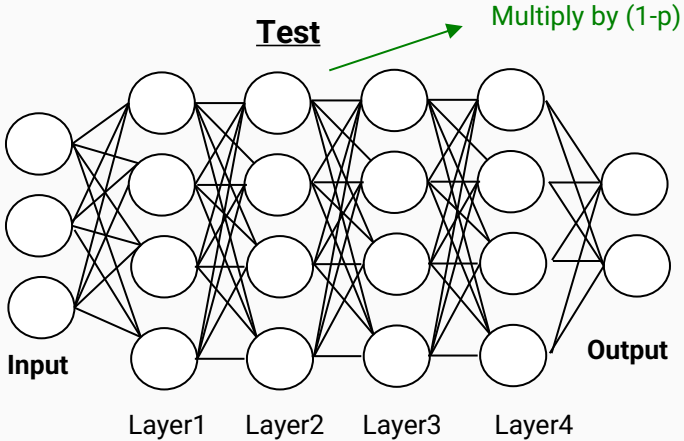




- Let's assume we have the following inputs to a certain neuron: [0.5, 0.5, 0.,5 0.5. 0.5].
- If **p=0.20,** on average 20% of these inputs will be dropped. After applying dropout, I can have the following input: [0.5, 0.5, 0.,5 **0.0**. 0.5].
- The neuron performs a weighted sum of these inputs (let's assume that all weights are 1 for simplicity).
- The sum of the inputs is **2.0.**
- At test time, we do not apply dropout and we have all the neurons active. In this case, the sum will be **2.5.**

Dropout





- Sum with dropout (training): **2.0**
- Sum without dropout (test/inference): **2.5**

- We have a **mismatch** in the input scale between training and test! This might significantly harm the performance.

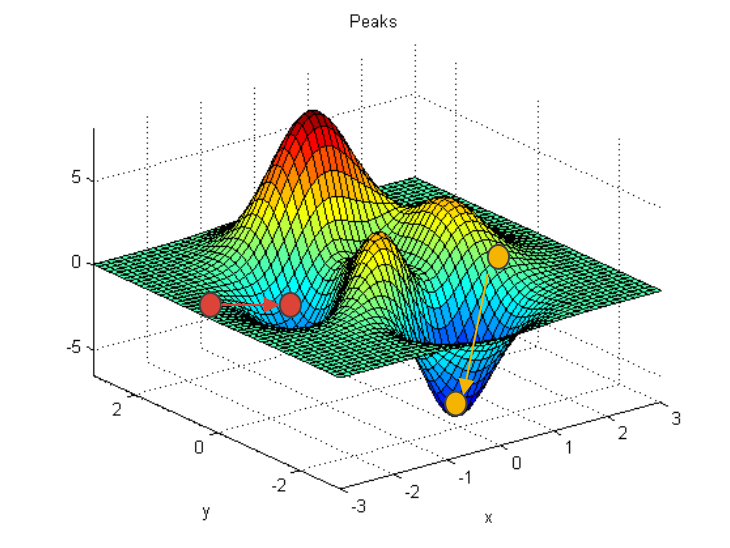
- During test, we can **compensate** for the effect of dropout by multiplying each input by **(1-p)**. In this case:
  0.8 * [0.5, 0.5, 0.5, 0.5, 0.5].

- The sum will be 2.0 (which is the same expected during training).

Initialization

Initialization

- Neural networks are trained with **gradient descent**.



- Initializing the network with **different weights** corresponds to a **different starting point** in the parameter space.

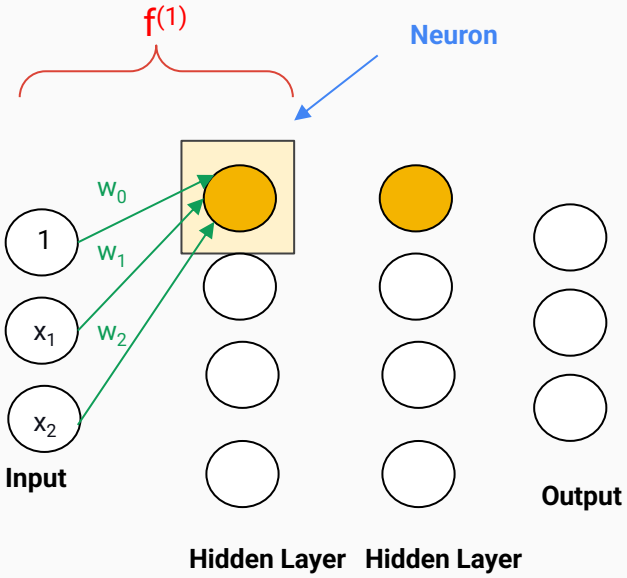
- This leads to **different final sets of weights** with possible **different performance**.

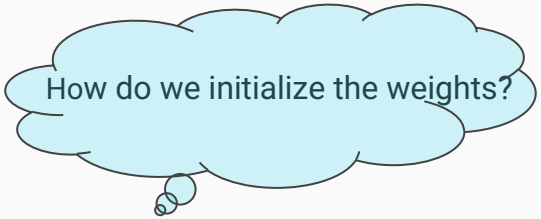
- The initial point can determine where the algorithm **converges**.

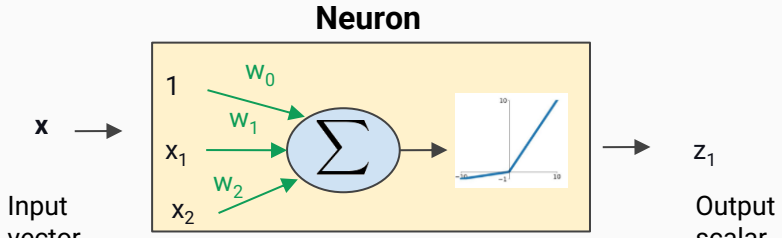
- Some initial points can be so **unstable** that the algorithm encounters **numerical issues** and fails.

Initialization

- Defining a **proper magnitude** and **range** of **variability** for the weights helps **speed up training** and **avoid numerical instabilities**.



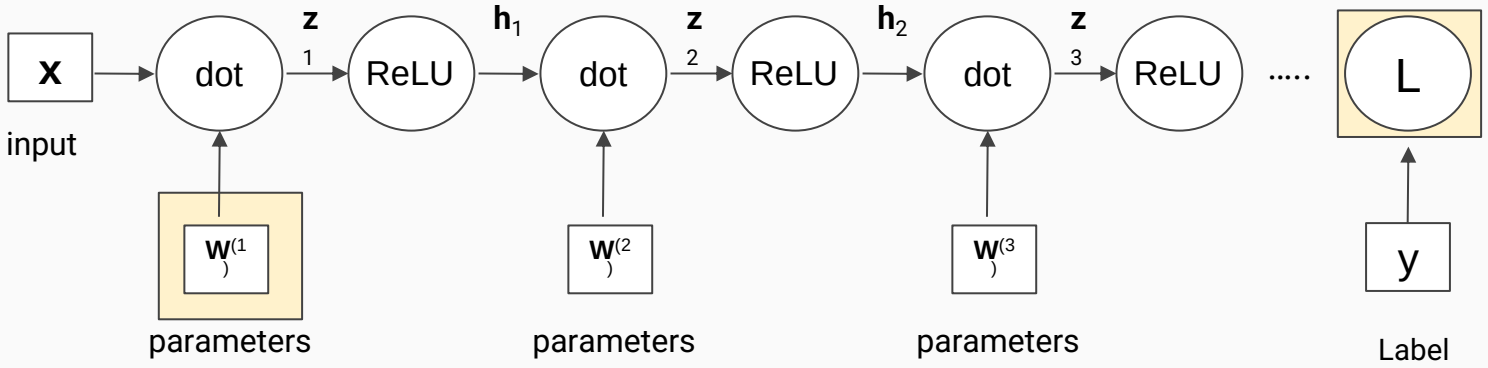




$$z_i = f^{(1)}\left(\sum_{j=0}^{D} w_{ji}^{(1)} x_j\right)$$

Initialization

- **Initializing** the **weights** properly is important to **avoid vanishing** and **exploding gradients**.



$$\frac{\partial L}{\partial w_1} = x\ step(z_1)\ w^{(2)} step(z_2) w^{(3)} step(z_3) ... w^{(L)} (\sigma(z_L) - y)$$

$$= x(\sigma(z_L) - y) \cdot \prod_{i=1}^{L-1} step(z_l) \cdot \prod_{i=2}^{L} w^{(i)}$$

We multiply the weight matrices multiple times

Initialization

$$\frac{\partial L}{\partial w_1} = x\ step(z_1)\ w^{(2)} step(z_2) w^{(3)} step(z_3) ... w^{(L)} (\sigma(z_L) - y)$$

$$= x(\sigma(z_L) - y) \cdot \prod_{i=1}^{L-1} step(z_l) \cdot \prod_{i=2}^{L} w^{(i)}$$

We multiply the weights multiple times

- If the weights have a large magnitude, we have **exploding gradients**.
- If the weights have a small magnitude, we have **vanishing gradients**.

- We have to **properly initialize** the weights to avoid these issues.

For simplicity, here we are considering a single (scalar) weight for each layer. When weights are matrices, the product becomes **dot products** between **matrices**.

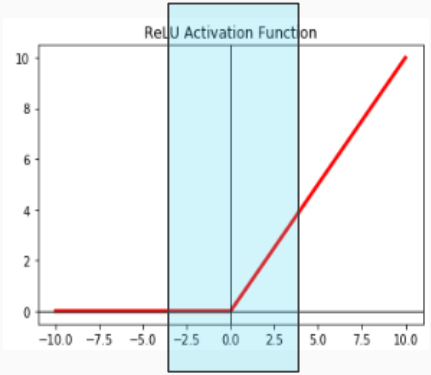
Initialization

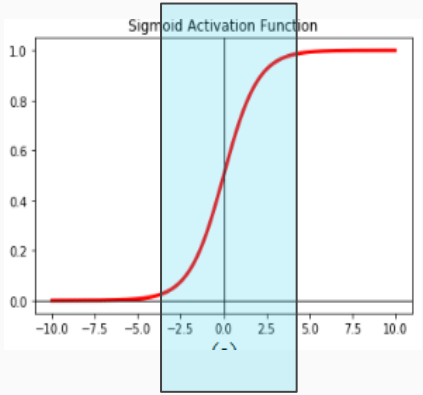
$$\frac{\partial L}{\partial w_1} = x\ step(z_1)\ w^{(2)} step(z_2) w^{(3)} step(z_3) \ldots w^{(L)} (\sigma(z_L) - y)$$

$$= x(\sigma(z_L) - y) \cdot \prod_{i=1}^{L-1} step(z_l) \cdot \prod_{i=2}^{L} w^{(i)}$$

We multiply the weight matrices multiple times

- To derive a good initialization scheme, we have to consider the **activation functions** as well.



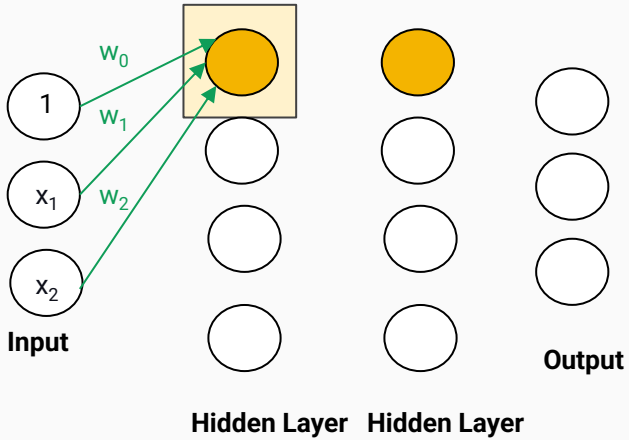


Intuitively, we want all neurons of all the layers to work around “zero” within a reasonable range of variability.

Ideally, we want all the neurons to have the same variance.

Initialization

- Based on this principle, some initialization schemes are proposed in the literature



Historically, weight initialization has been done with **simple heuristics**, such as:

$$\mathbf{W}^{(l)} \sim -U\Big[-0.3, 0.3\Big]$$

More tailored approaches have been developed over the last decade (that have become the de facto standard):

- **Glorot's initialization** (for tanh activations)

$$\mathbf{W}^{(l)} \sim -U\Big[-\frac{\sqrt{6}}{\sqrt{n^{(l)}+n^{(l+1)}}}, \frac{\sqrt{6}}{\sqrt{n^{(l)}+n^{(l+1)}}}\Big]$$

Number of neurons in the layer l

- **He initialization** (for ReLU activations)

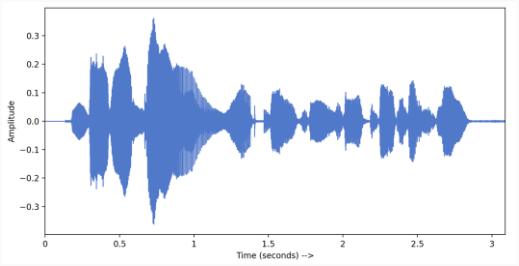
$$\mathbf{W}^{(l)} \sim \mathcal{N}(\mu = 0, \sigma^2 = \frac{2}{n^{(l)}})$$

Recurrent Neural Networks

Recurrent Neural Networks

- Many real-life data are actually **sequences** (e.g., time series):

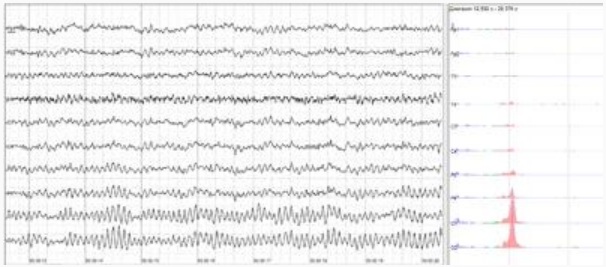
Audio



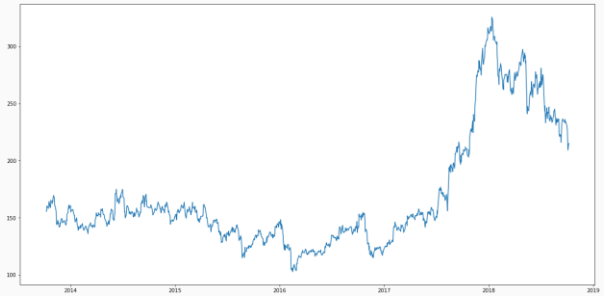
Video

.....

EEG Brain signals

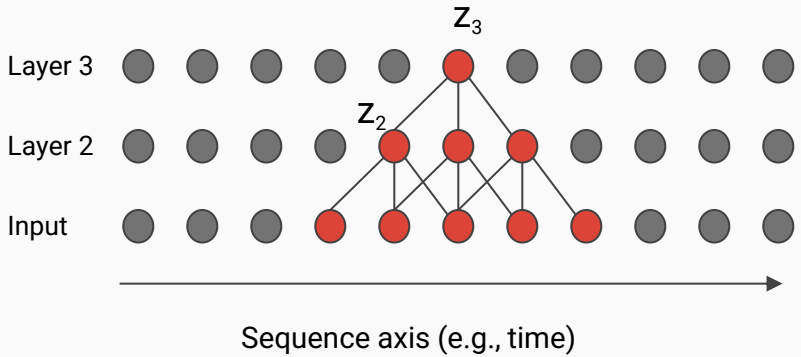


Stock Price Prediction



Recurrent Neural Networks

- In the last lecture, we have seen that we can process sequences with 1d **convolutional neural networks**:



- CNNs, however, are designed to **capture local dependencies** (i.e, the ones available in the receptive field).

- For some types of signals, we want to learn **long-term dependencies**.

- For instance, we might want to have a model whose neurons depend on the **full history** (i.e., **all** the **past elements**).

Recurrent Neural Networks

- In **Recurrent Neural Networks** (RNNs), the **current output** depends on **all the previous inputs**.
- The general form for an RNN is the following:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W})$$

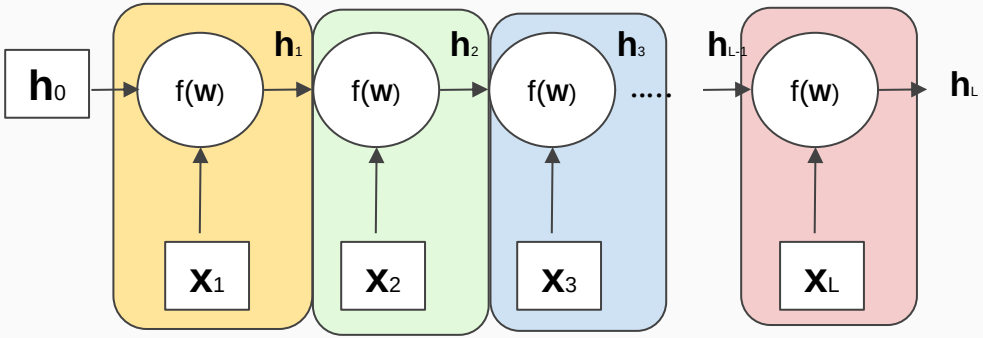
The **current state $\mathbf{h}_t$** depends on:

- The current input **$\mathbf{x}_t$**
- The previous state **$\mathbf{h}_t$**
- A set of learnable parameters **W**

Recurrent Neural Networks

- We can **unfold** the recurrent neural networks in this way:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W})$$



$$\mathbf{h}_0 = 0$$

$$\mathbf{h}_1 = f(\mathbf{x}_1, \mathbf{h}_0, \mathbf{W})$$

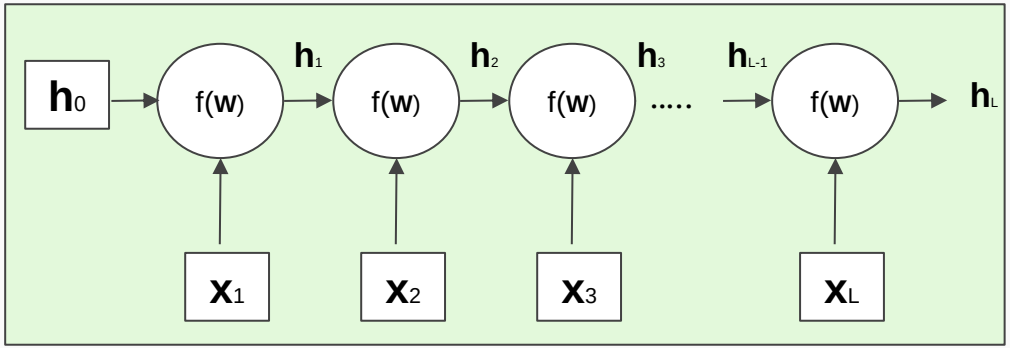
$$\mathbf{h}_2 = f(\mathbf{x}_2, \mathbf{h}_1, \mathbf{W})$$

$$\mathbf{h}_3 = f(\mathbf{x}_3, \mathbf{h}_2, \mathbf{W})$$

$$\mathbf{h}_L = f(\mathbf{x}_L, \mathbf{h}_{L-1}, \mathbf{W})$$

- The current state $\mathbf{h}_t$ depends on the current input $\mathbf{x}_t$ and all the previous ones.
- The final state $\mathbf{h}_L$ depends on **all the inputs.**

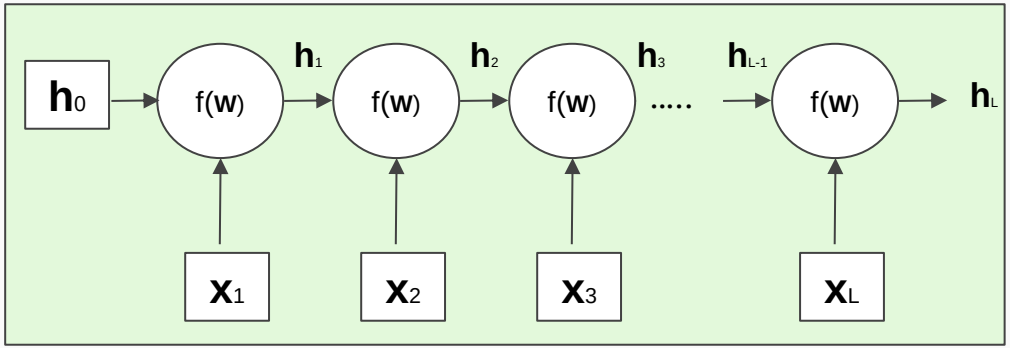
Recurrent Neural Networks



$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W})$$

- We apply the same neural network *f* for **all the sequence steps** (**weight sharing**).
- As seen for CNNs, applying the **same weights** over the input helps **find patterns** in the data.
- In CNNs, we share the same filter over the inputs, for RNNs we share a **full neural network**.
- Moreover, RNNs we can find **arbitrary long patterns** because the state vector acts as a memory of the previous inputs.

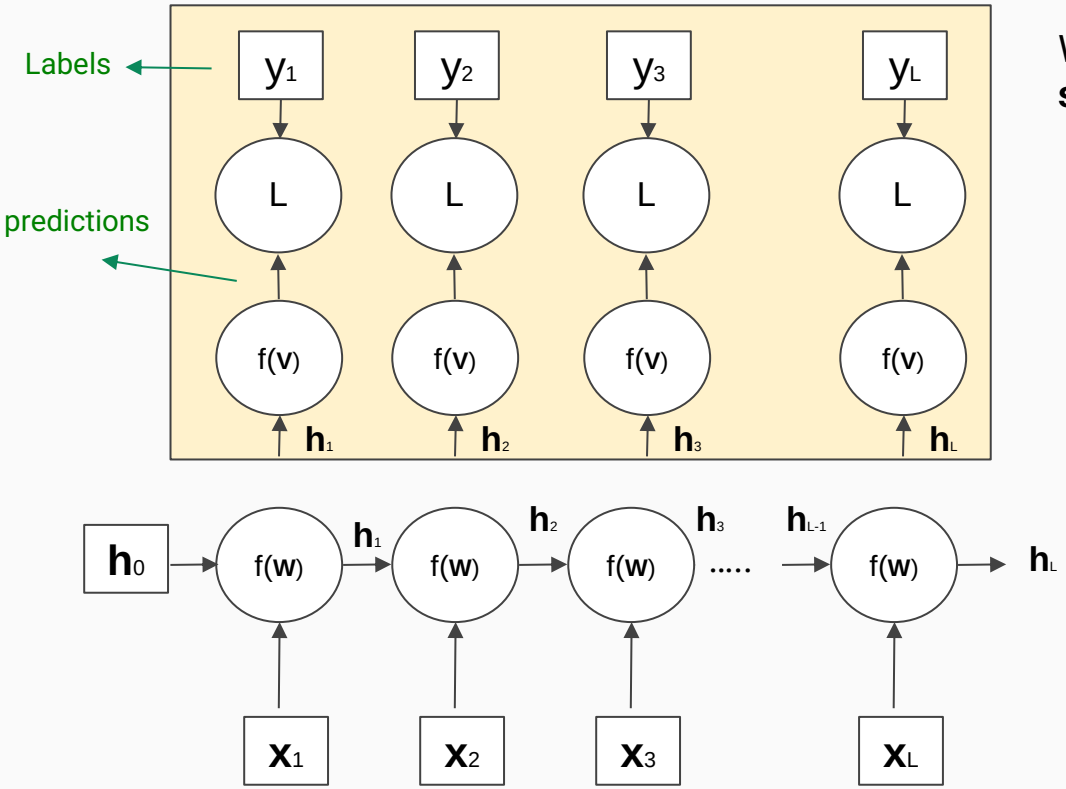
Recurrent Neural Networks



$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}, \mathbf{W})$$

- Once unfolded, RNN can be represented as a single **big computational graph**.
- We can see a recurrent neural network as a **deep neural network**, which is deep in the **time axis**.
- RNNs can be used for **different types** of **problems,** that include:
  - Problems where we need a **prediction at each time step** (Many-to-many).
  - Problems where we need a single **prediction at the end** (Many-to-one).
  - Problems where we want to turn the **input sequence** of length T into an **output sequence of** length S **(Seq-to-seq)**.

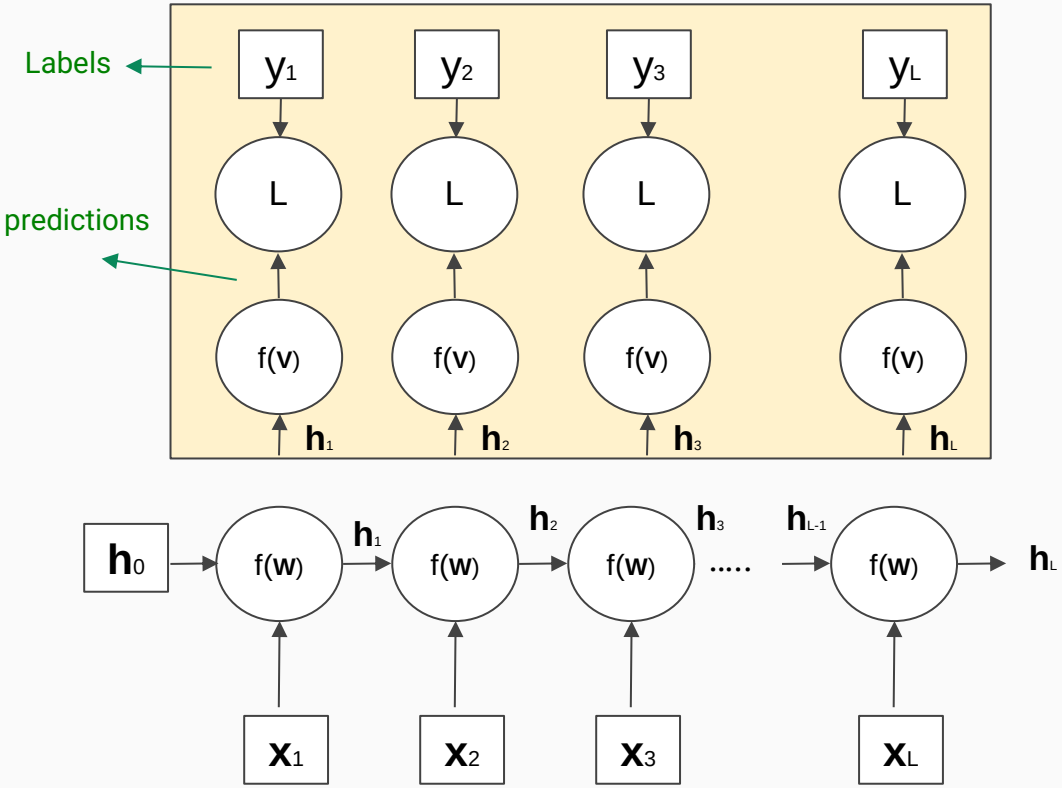
Many-to-Many Problems



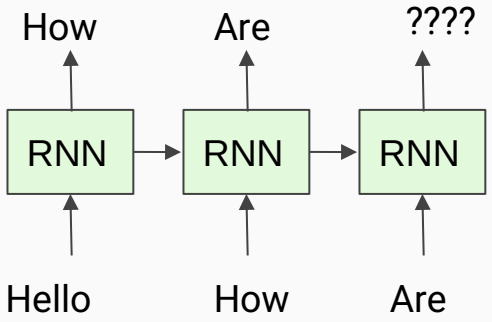
When we need a **single prediction at each time step** (many-to-many problem), we have to:

- Apply a **final transformation** (e.g., linear model) on top of each hidden state.
- Apply a **loss function at each time step** (e.g., Categorical Cross-Entropy for multi-class classification, MSE for regression).
- The **total loss** will be the **sum** (or **average**) of all the losses at each time step.

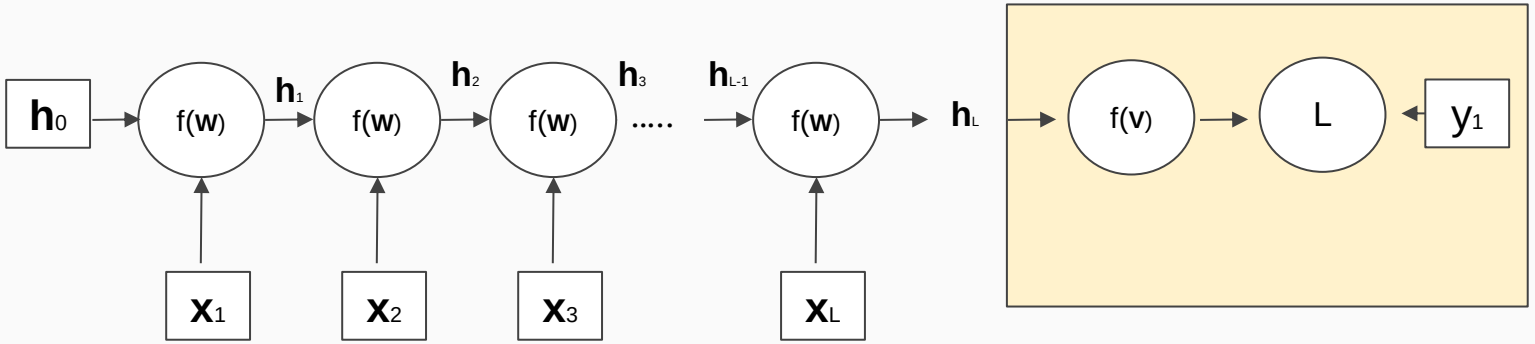
Many-to-Many Problems



- Examples of problems like this are **stock predictions**, where we need to predict a value every day or hour.
- Another example is **language modeling**, where we want to predict the next word given the previous ones.



Many-to-one Problems



When we need a **single prediction** in output (many-to-one problem), we can do the following:

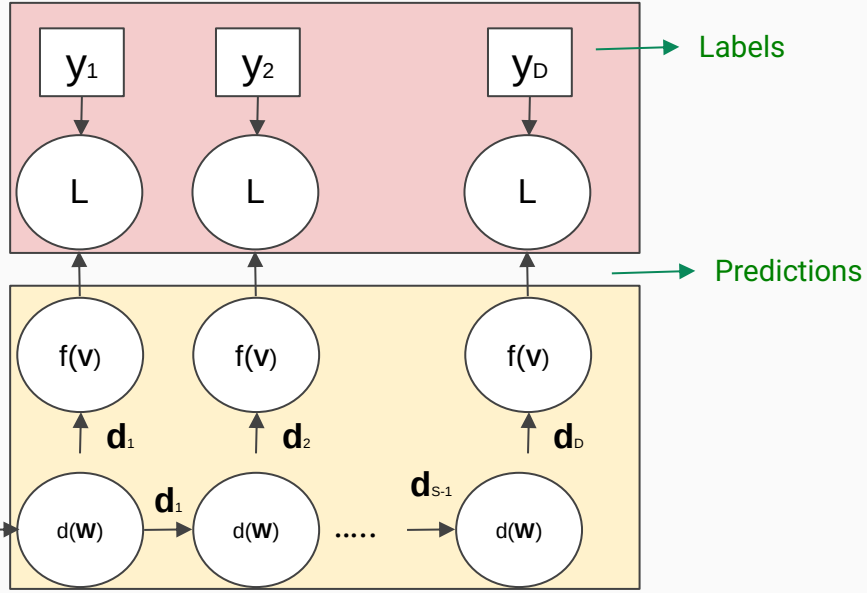
- Apply a **final transformation** (e.g., linear model) on top of the **last hidden state** (which depends on all the inputs).
- Apply a **loss function** on top of it (e.g., Categorical Cross-Entropy for multi-class classification, MSE for regression).

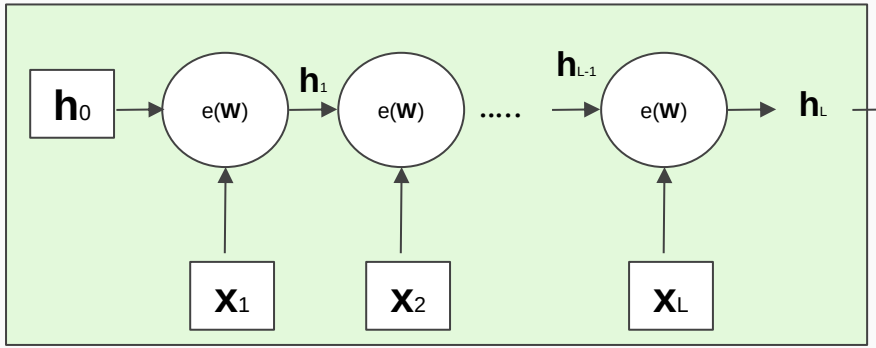
Examples of problems like this are **speaker identification,** emotion recognition from text, etc.

Sequence-to-Sequence Problems

- RNNs can also be used for **sequence-to-sequence problems**:
- We employ an RNN-based **encoder** that encodes all the L inputs.
- We employ an RNN-based **decoder** that takes one of many encoded states and generates (one-by-one) the output elements.
- A **loss** is computed on top of **each prediction.**

Attention mechanisms are often used to connect encoder and decoder states.

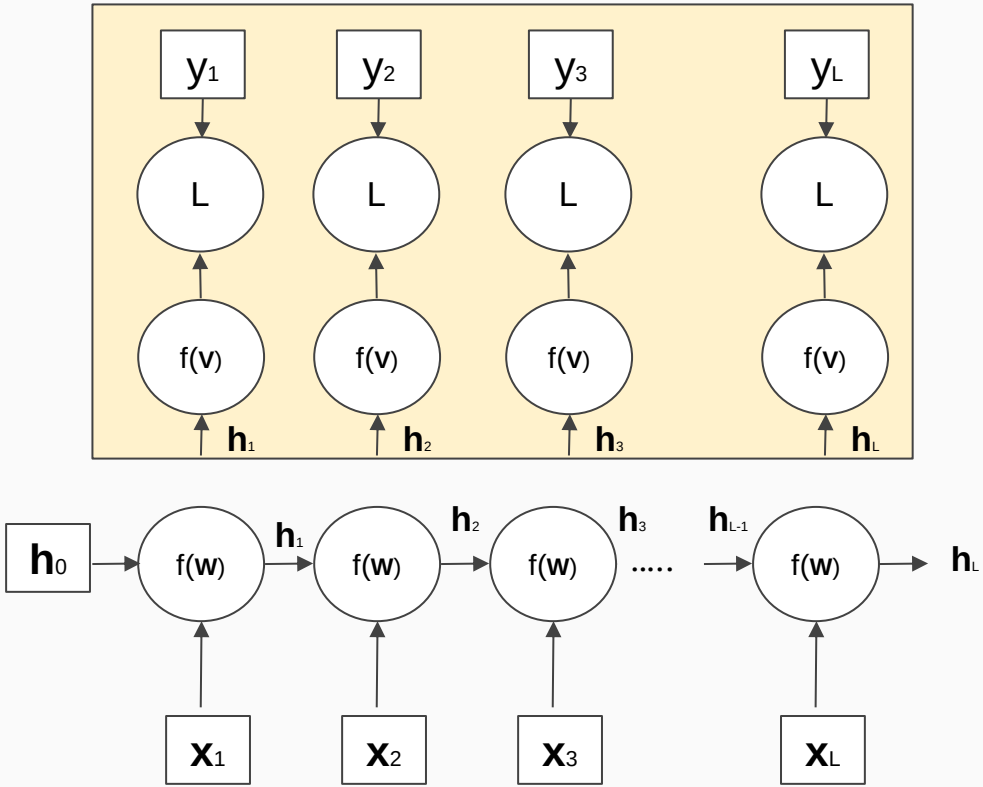




- Examples of seq2seq problems are **machine translation** and **speech recognition**.

Training

- *How can we train an RNN?*



- After unfolding it, the RNN becomes a standard **computational graph** that employs long chains of computations.
- We can thus compute the gradient with the **backpropagation algorithm**.
- To highlight that the gradient is mainly propagated over time, it is sometimes called **backpropagation through time**.
- Once we have the gradient, we can update the parameters with **gradient descent** (and all its variants seen so far).

Vanilla RNN

- The simplest RNN is called **Vanilla RNN** (or Elman RNN) is based on the following equation:

$$\mathbf{h}_t = tanh(\mathbf{W}^{(in)T}\mathbf{x}_t + \mathbf{W}^{(hh)T}\mathbf{h}_{t-1})$$

We perform a **linear transformation** of both the current input and the previous state.

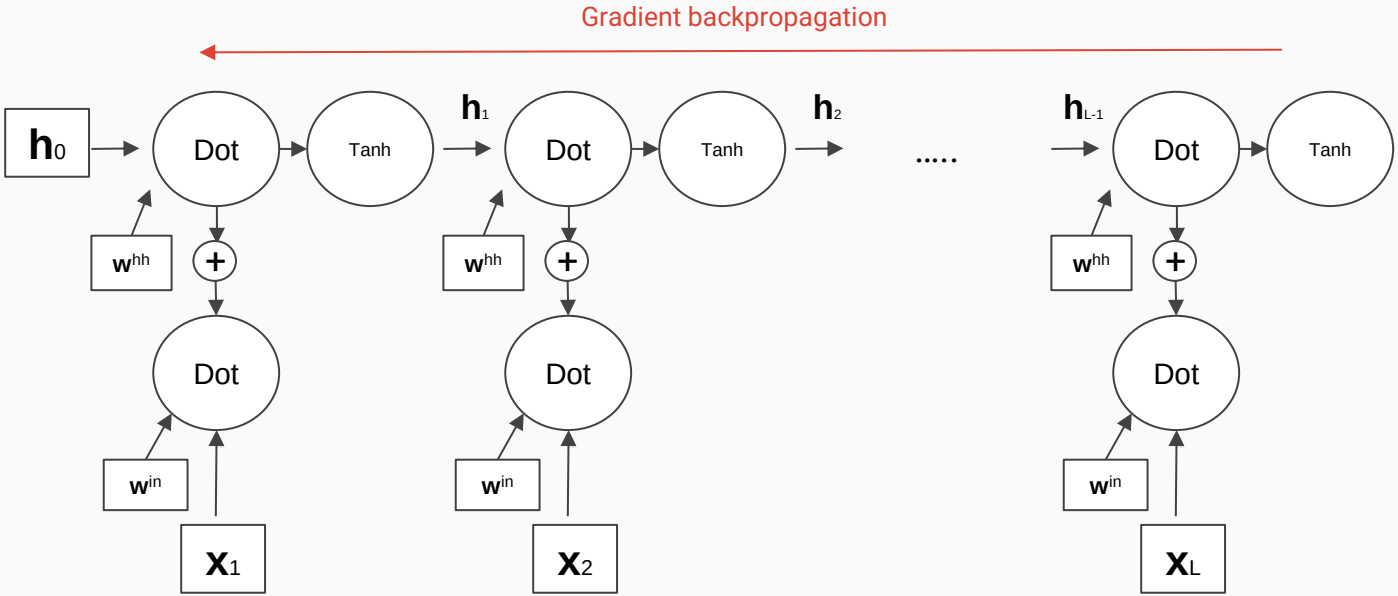
We sum them up and apply a **non-linearity** (Tanh or ReLU).

$$\mathbf{W}^{(in)} = \begin{bmatrix} w_{0,1}^{(in)} & w_{0,2}^{(in)} & \dots & w_{0,M}^{(in)} \\ w_{1,1}^{(in)} & w_{1,2}^{(in)} & \dots & w_{1,M}^{(in)} \\ \dots & \dots & \dots & \dots \\ w_{D,1}^{(in)} & w_{D,2}^{(in)} & \dots & w_{D,M}^{(in)} \end{bmatrix} \quad \mathbf{x}_t = [1, x_{t,1}, x_{t,2}, \dots, x_{t,D}]^T$$

$$\mathbf{W}^{(hh)} = \begin{bmatrix} w_{1,1}^{(hh)} & w_{1,2}^{(hh)} & \dots & w_{1,M}^{(hh)} \\ w_{2,1}^{(hh)} & w_{2,2}^{(hh)} & \dots & w_{2,M}^{(hh)} \\ \dots & \dots & \dots & \dots \\ w_{M,1}^{(hh)} & w_{M,2}^{(hh)} & \dots & w_{M,M}^{(hh)} \end{bmatrix} \quad \mathbf{h}_t = [h_{t,1}, h_{t,2}, \dots, h_{t,M}]^T$$

Vanishing Gradient

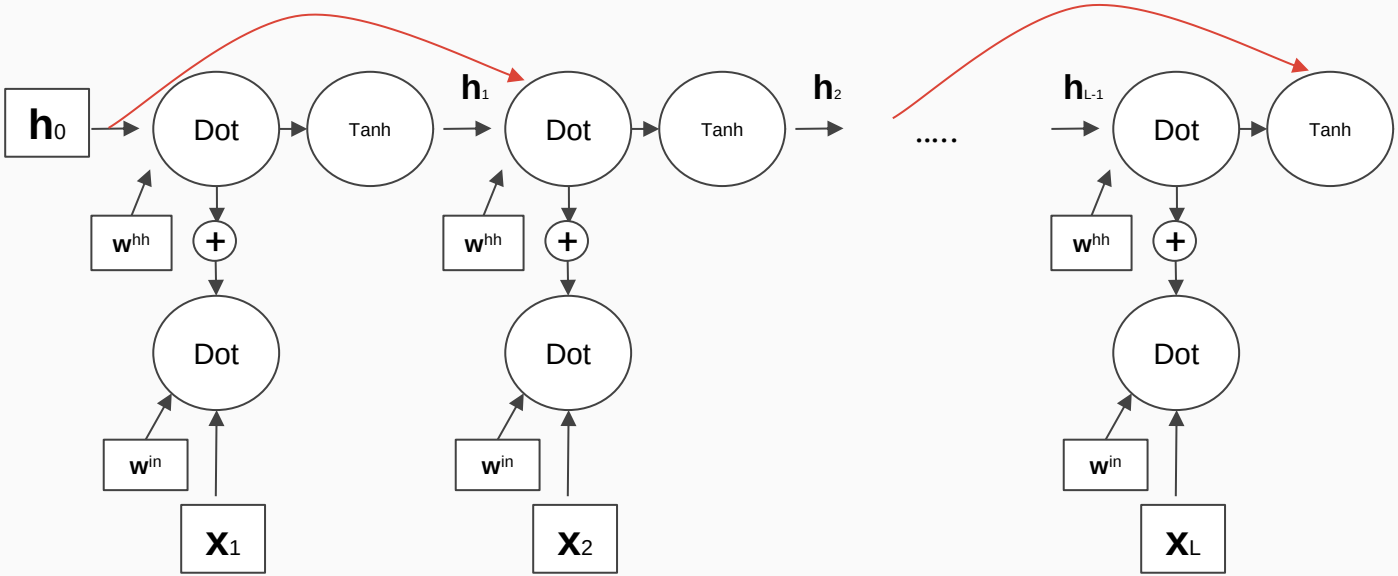
- When using RNNs, we have to backpropagate the gradient through a **long temporal** chain.
- As we have seen, this might cause **exploding gradients** or **vanishing gradients**.
- The vanishing gradient problem prevents the model to learn **long-term dependencies**.



Vanishing Gradient

- As we have seen in the previous lecture, we can add **shortcuts** in the architecture to fight vanishing gradients.

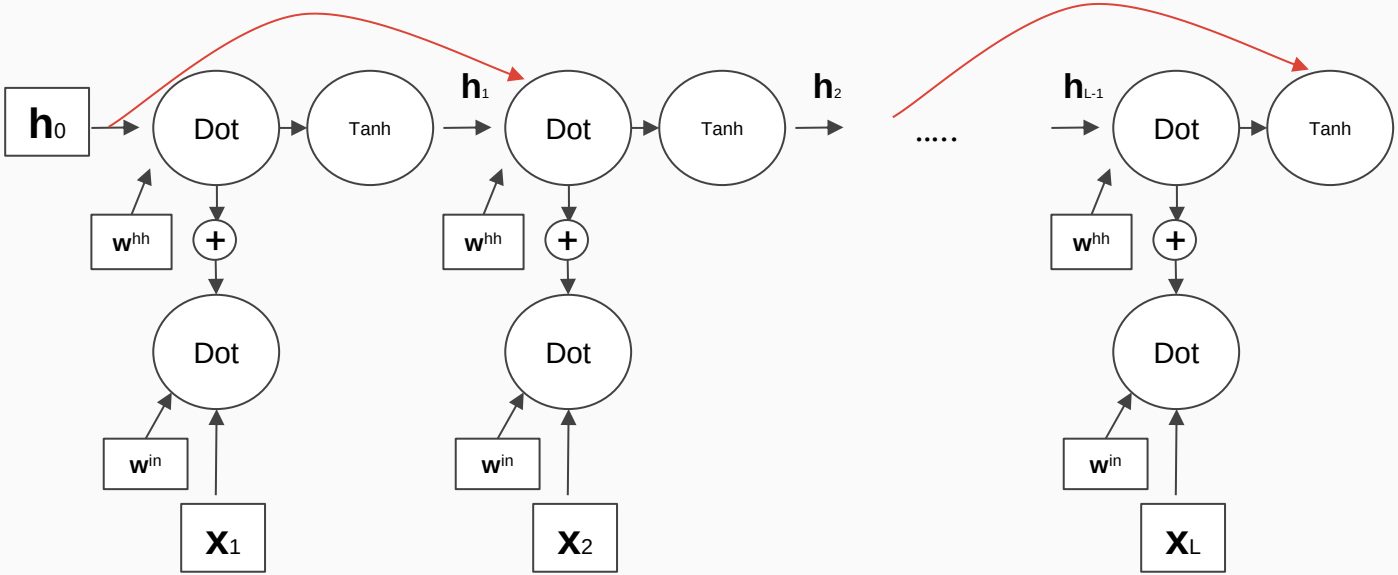
- In this case, we can consider shortcuts across **time steps**.



Vanishing Gradient

Instead of **hard-coding pre-defined shortcuts**, why don't we try to **learn** them?

Ideally, we want to learn **"dynamic" shortcuts** that connect relevant time steps.

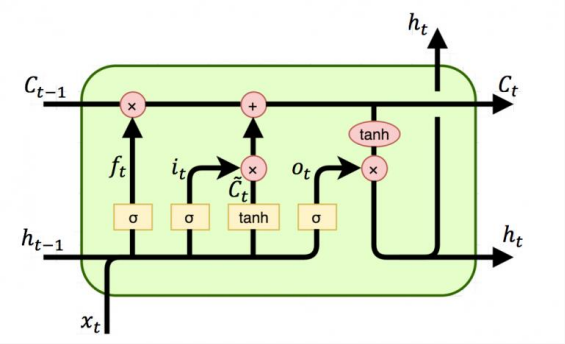


Gated RNNs

- This is exactly what we are trying to do with **multiplicative gates**:

Long Short-Term Memory (LSTM)

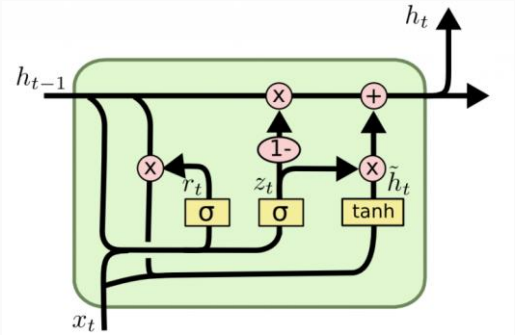
$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$



3 multiplicative gates

Gated Recurrent Units (GRU)

$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
\hat{h}_t &= \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t
\end{aligned}
$$



2 multiplicative gates

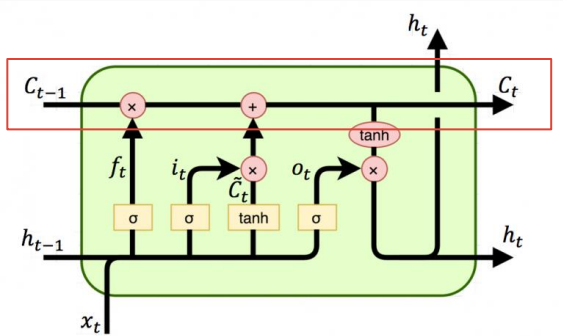
Light Gated Recurrent Units (Li-GRU)

$$
\begin{aligned}
z_t &= \sigma(BN(W_z x_t) + U_z h_{t-1}) \\
\widetilde{h}_t &= \mathrm{ReLU}(BN(W_h x_t) + U_h h_{t-1}) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t
\end{aligned}
$$

If the update gate z is 1, we can remember forever the state h (thus learning arbitrary **long-term dependencies**)

1 multiplicative gate

LSTM



- LSTM manages the flow of information using three **multiplicative gates** (called input, output, and forget gates).
- The cell state is the **memory vector**.
- It's very easy for information to just flow along it **unchanged**.

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o)
\end{aligned}
$$

→ Multiplicative Gates

$$
\begin{aligned}
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t
\end{aligned}
$$

→ Cell state

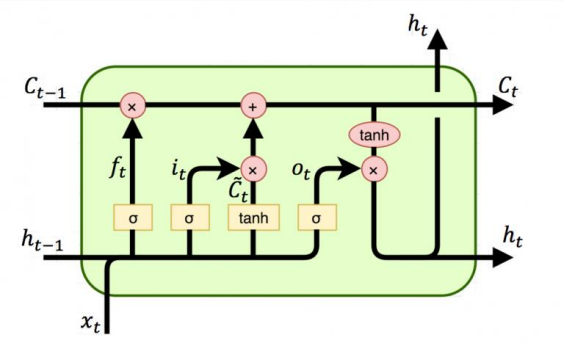
$$
h_t = o_t \circ \sigma_h(c_t)
$$

- To remember the past cell state, we just need to set the forget to one and the input gate to zero.
- This implements the "**shortcut**" as we can skip time steps and store information for an **arbitrarily large** number of steps.

Hochreiter & Schmidhuber (1995)

https://www.bioinf.jku.at/publications/older/2604.pdf

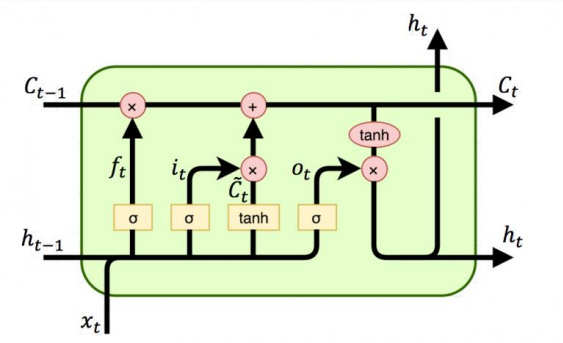
LSTM



$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ \sigma_h(c_t)$$

- More specifically, the role of the gates is the following:
  - **Input Gate**: It decides what new information we're going to store in the cell state.
  - The outcome is a vector containing numbers between 0 and 1 (due to the sigmoid activation):
  - "0" means" do not store the content of the current input".
  - "1" means" store the content of the current input unchanged".

LSTM



$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$
$$h_t = o_t \circ \sigma_h(c_t)$$

- **Forget Gate**: It decides how much information keep from the previous cell state.
- “0” means“ totally forget the past.
- “1” means” keep the past information unchanged.
- **Output Gate**: It decides how much information from the cell stare expose in the output
- “0” means“ do not expose the cell state
- “1” means” expose the full cell state

Light-GRU

https://arxiv.org/abs/1803.10225

- A further simplification is the Light Gated Recurrent Unit (Li-GRU)

- It only uses 1 multiplicative gate:

$$z_t = \sigma(BN(W_z x_t) + U_z h_{t-1})$$
$$\widetilde{h_t} = \mathrm{ReLU}(BN(W_h x_t) + U_h h_{t-1})$$
$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h_t}$$

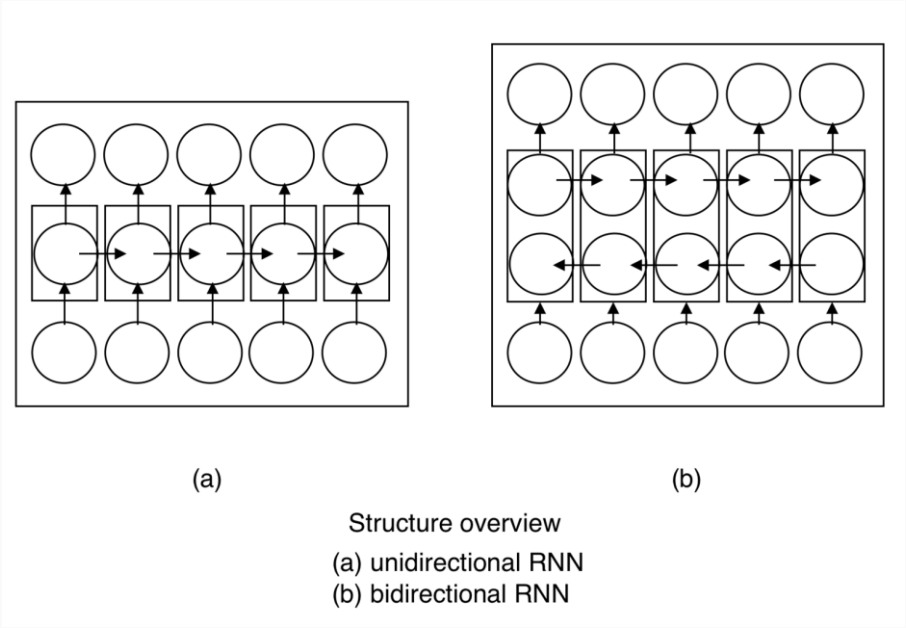
- **Update Gate**: It decides how much information from the past to use for the current prediction and store for the future one.

- There are other improvements: ReLU + Batchnorm is used.

- Li-GRU is faster and performs better than GRU in speech processing tasks.

Bidirectional RNN

- In some cases, we want to make a prediction at each time step based on the **whole input elements** and not only the **previous ones**.



Structure overview
(a) unidirectional RNN
(b) bidirectional RNN

We can employ two RNNs running in **opposite directions**.

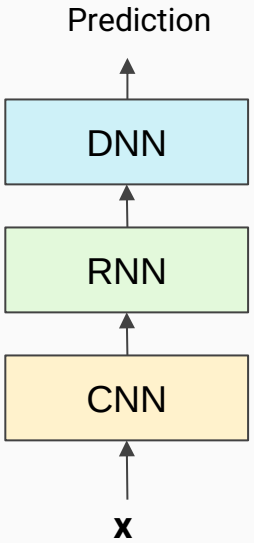
These RNNs use different parameters or share them.

At each layer, we can **combine the forward and the backward state** by concatenating them or summing them up.

Examples of applications are *speech recognition*, *machine translation*, and *handwritten recognition*.

CNNs + RNNs + MLPs

- We can make RNNs even deeper by **stacking multiple RNN layers**.
- We can also combine CNNs, RNNs, and DNNs (MLPs):



This model is called CRDNN and is very powerful:

- It first learns **local contexts** with a **CNN**.
- It then captures **long-term dependencies** with an **RNN**.
- Finally, it performs a **final classification** with an MLP.

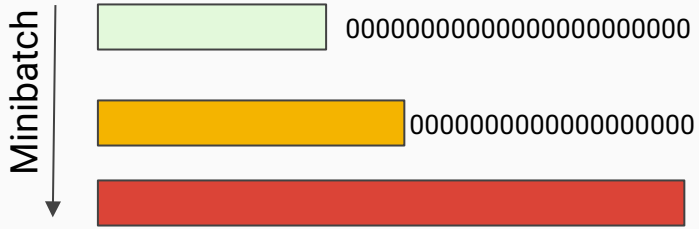
All the blocks are **jointly trained**.

- This is one of the models used in SpeechBrain (https://speechbrain.github.io/) to process speech signals.

Variable-Length Sequences

- Often, the sequences in input to the RNN have **different lengths**.
- For instance, think about a speech signal: in some cases, we might have long recordings, and in some others short ones.
- H*ow can we manage variable-length sequences?*

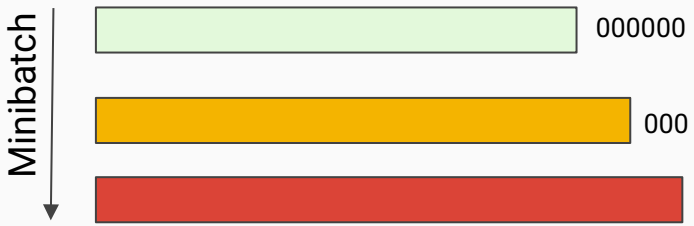
We can handle it by **zero-padding:** within each minibatch, we pad with zeros the inputs to match the length of the longest one **.**



If the length of the inputs is very different, this is **computationally inefficient** because we waste time processing zeros.

Variable-Length Sequences

- One way to mitigate this issue is to **sort** the inputs by length (in ascending or descending order) before creating the minibatches.

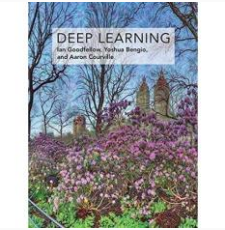


This minimizes the need for zero-padding, but **sacrifices randomness** in the minibatch creation:

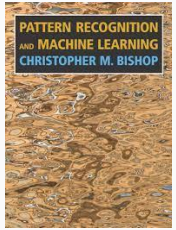
since we sorted the data, minibatches composed of the same data are shown across epochs.

- A compromise solution can be implemented through **bucketing**:
- We split data into **buckets** such that each bucket contains **inputs** of **similar lengths**.
- When creating minibatches, we sample random inputs from the **same bucket**.

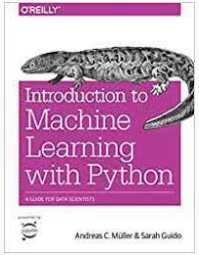
Additional Material



Chapter 2: Linear Algebra
Chapter 3: Probability and information theory
Chapter 5: Machine Learning Basics
Chapter 6: Deep Feedforward Networks
Chapter 9: Convolutional Networks
Chapter 10: **Sequence Modeling**



1.1.0 Example: Polynomial Curve Fitting
1.2.0 Probability Theory
3.1.0 Linear Basis Function Models
3.1.1 Maximum likelihood and least squares
4.3.2 Logistic regression
5.0 - 5.4 Neural Networks
5.5.6: Convolutional Networks



Introduction (page 1-27)
Linear Models (page 47-70)
Neural Network (page 106-121)

Lab session

- During the weekly lab session, we will do:

colab Tutorial on GPU computing.

colab Tutorial on Vanishing Gradient.

Recurrent Neural Networks

Lab Assignment