Introduction to Deep Neural Networks (2)

Piyush Rai

Introduction to Machine Learning (CS771A)

October 25, 2018

Plan for today

- Quick recap of feedforward networks
- Backprop via a small example
- Variations/improvements to basic feedforward networks
  - Convolutional Neural Networks (CNN)
  - Neural Networks for sequential data (RNN and LSTM)
- Neural networks for unsupervised learning (deep autoencoders)
- Some other recent advances (GAN and VAE)

Plan for today

- Quick recap of feedforward networks
- Backprop via a small example
- Variations/improvements to basic feedforward networks
  - Convolutional Neural Networks (CNN)
  - Neural Networks for sequential data (RNN and LSTM)
- Neural networks for unsupervised learning (deep autoencoders)
- Some other recent advances (GAN and VAE)
- Note: The attempt (this as well as previous lecture) is to convey basic principles of deep neural networks. For a more in-depth treatment, you are advised to take a dedicated deep learning course

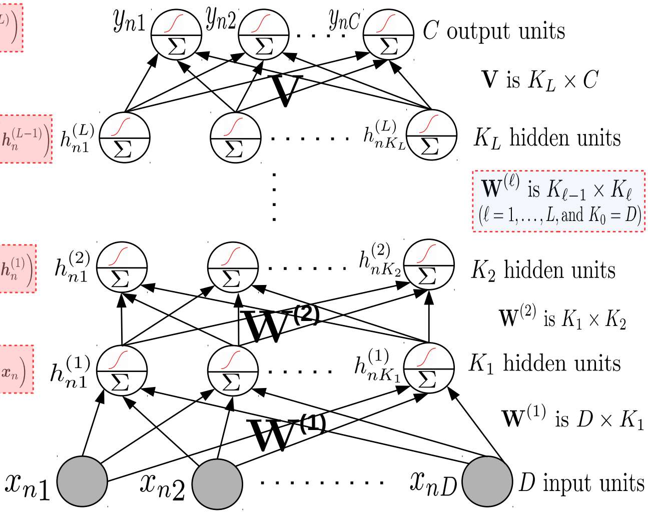
Recap: Feedforward Neural Networks (MLP)

$y_n = o\left(\mathbf{V}^\top h_n^{(L)}\right)$

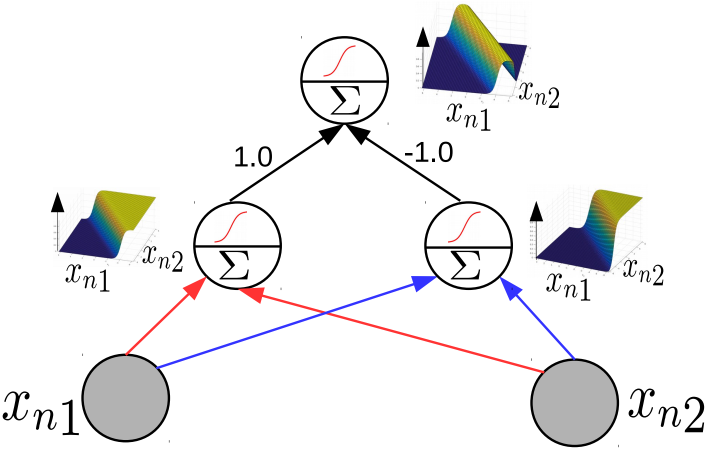
$h_n^{(L)} = g\left({\mathbf{W}^{(L)}}^\top h_n^{(L-1)}\right)$

$h_n^{(2)} = g\left({\mathbf{W}^{(2)}}^\top h_n^{(1)}\right)$

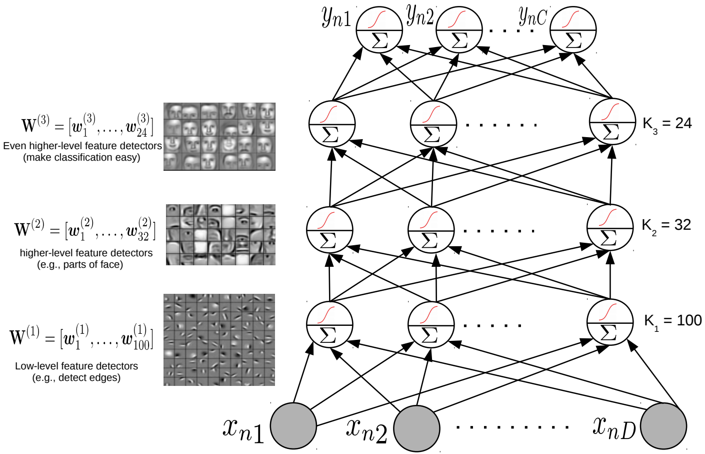
$h_n^{(1)} = g\left({\mathbf{W}^{(1)}}^\top x_n\right)$



Recap: MLP as Composition of Functions



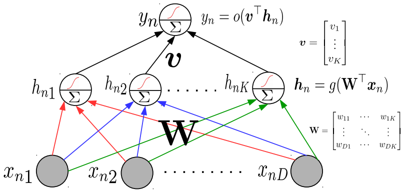
Recap: MLP as Multi-layer Feature Detector



Note: If no. of hidden units $< D$, then it can also be seen as doing (supervised) dim-red

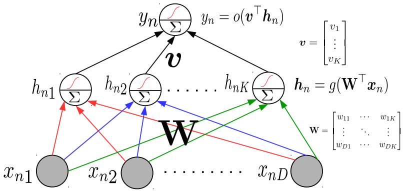
Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP

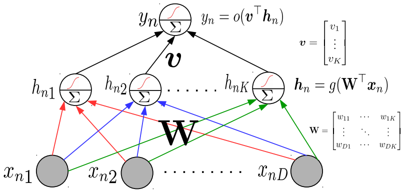


- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



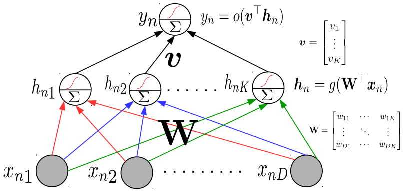
- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP

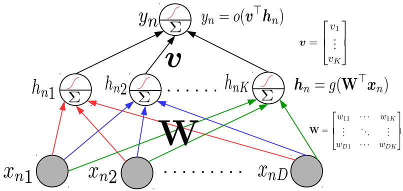


- Assuming regression (o = identity), the loss function for this model

$$\begin{aligned}
\mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2
\end{aligned}$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

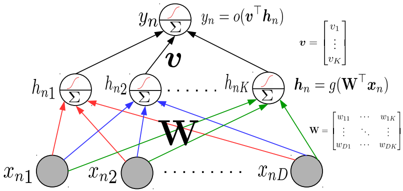
$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

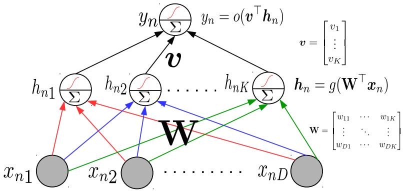
$$\begin{aligned}
\mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2
\end{aligned}$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk}$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

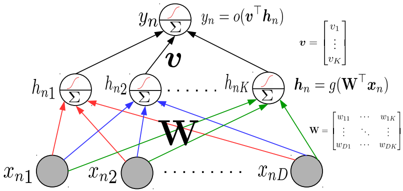
$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

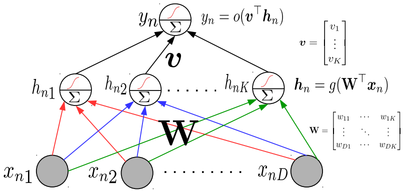
$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression ($o =$ identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

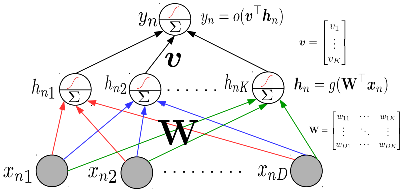
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$

$$\frac{\partial \mathcal{L}}{\partial h_{nk}} = -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

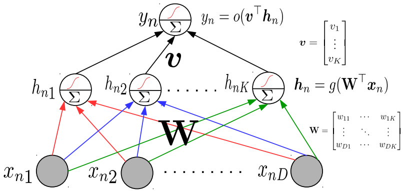
$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$

$$\frac{\partial \mathcal{L}}{\partial h_{nk}} = -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k$$

$$\frac{\partial h_{nk}}{\partial w_{dk}} = g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad \text{(note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))$$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2 \\
&= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2
\end{aligned}
$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$
\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}
$$

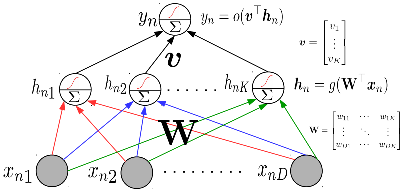
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{dk}} &= \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}} \\
\frac{\partial \mathcal{L}}{\partial h_{nk}} &= -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k \\
\frac{\partial h_{nk}}{\partial w_{dk}} &= g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad (\text{note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))
\end{aligned}
$$

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$
$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$
$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

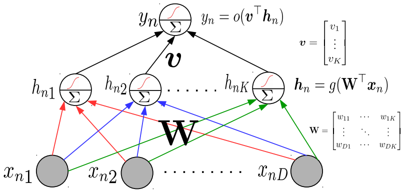
- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$
$$\frac{\partial \mathcal{L}}{\partial h_{nk}} = -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k$$
$$\frac{\partial h_{nk}}{\partial w_{dk}} = g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad (\text{note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))$$

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$. Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods

Learning MLP via Backpropagation: A Simple Example

- Consider a single hidden layer MLP



- Assuming regression (o = identity), the loss function for this model

$$\mathcal{L} = \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \boldsymbol{v}^\top \boldsymbol{h}_n\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k h_{nk}\right)^2$$

$$= \frac{1}{2}\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right)^2$$

- To use gradient methods for $\mathbf{W}$, $\boldsymbol{v}$, we need gradients.
- Gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{v}$ is straightforward

$$\frac{\partial \mathcal{L}}{\partial v_k} = -\sum_{n=1}^{N}\left(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)\right) h_{nk} = \sum_{n=1}^{N} \boldsymbol{e}_n h_{nk}$$

- Gradient of $\mathcal{L}$ w.r.t. $\mathbf{W}$ requires chain rule

$$\frac{\partial \mathcal{L}}{\partial w_{dk}} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}}{\partial h_{nk}} \frac{\partial h_{nk}}{\partial w_{dk}}$$

$$\frac{\partial \mathcal{L}}{\partial h_{nk}} = -(y_n - \sum_{k=1}^{K} v_k g(\boldsymbol{w}_k^\top \boldsymbol{x}_n)) v_k = -\boldsymbol{e}_n v_k$$

$$\frac{\partial h_{nk}}{\partial w_{dk}} = g'(\boldsymbol{w}_k^\top \boldsymbol{x}_n) x_{nd} \qquad \text{(note: } h_{nk} = g(\boldsymbol{w}_k^\top \boldsymbol{x}_n))$$

- Forward prop computes errors $\boldsymbol{e}_n$ using current $\mathbf{W}$, $\boldsymbol{v}$. Backprop updates NN params $\mathbf{W}$, $\boldsymbol{v}$ using grad methods
- Backprop caches many of the calculations for reuse

Some Considerations w.r.t. Optimization in Deep NN

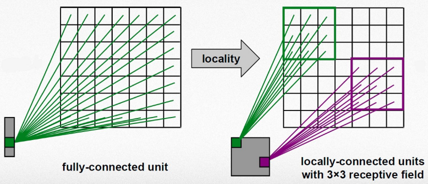
- Gradient based first-order methods are among the most popular ones
- Typically mini-batch SGD based method are used
- However, due to non-convexity, care needs to be exercised
  - Adaptive learning rates (Adam, Adagrad, RMSProp)
  - Momentum based or "look ahead" gradient methods

Some Considerations w.r.t. Optimization in Deep NN

- Gradient based first-order methods are among the most popular ones
- Typically mini-batch SGD based method are used
- However, due to non-convexity, care needs to be exercised
  - Adaptive learning rates (Adam, Adagrad, RMSProp)
  - Momentum based or "look ahead" gradient methods
- Initialization is also very important
  - Layer-wise pre-training was one of the first successful schemes
  - Many other heuristics exist now

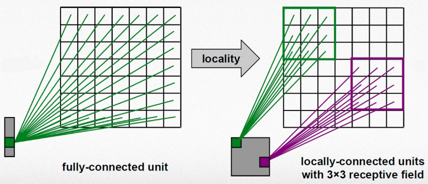
Some Limitations of Feedforward Networks

- Require a huge number of parameters (note that the consecutive layers are fully connected)
- Not ideal for data that exhibit locality structure, e.g., (e.g., images, sentences)
  - Kind of works but would be better to exploit locality in the data more explicitly

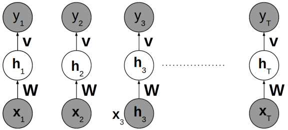


Some Limitations of Feedforward Networks

- Require a huge number of parameters (note that the consecutive layers are fully connected)
- Not ideal for data that exhibit locality structure, e.g., (e.g., images, sentences)
  - Kind of works but would be better to exploit locality in the data more explicitly

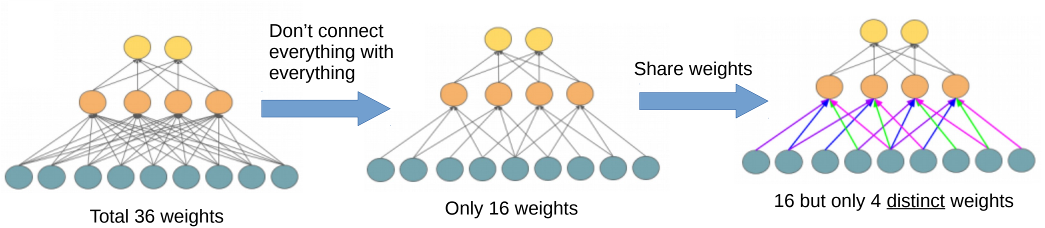


- Doesn't have a "memory", so not ideal when modeling sequence of observations



Convolutional Neural Network (CNN)

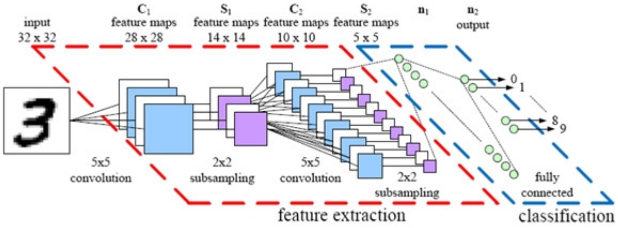
- A feedforward neural network with a special structure



- Not all pairs of nodes are connected
- Weights are also "tied" (many connections have the same weights; color-coded above)
- The set of distinct weights defines a "filter" or "local" feature detector

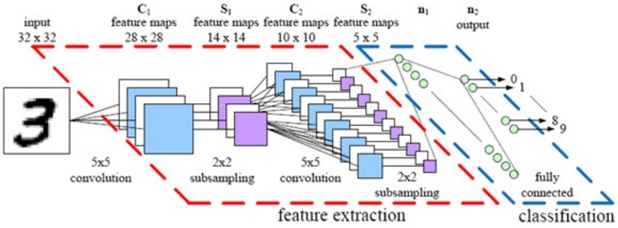
Convolutional Neural Network (CNN)

- Applies 2 operations, convolution and pooling (subsampling), repeatedly on the input data



Convolutional Neural Network (CNN)

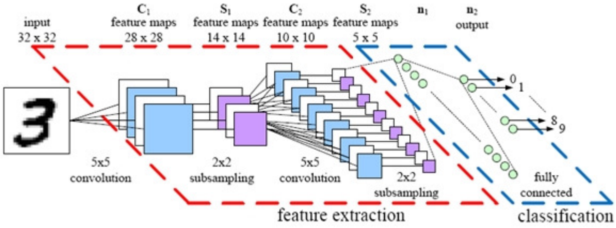
- Applies 2 operations, convolution and pooling (subsampling), repeatedly on the input data



- Convolution: Extract "local" properties of the signal. Uses a set of "filters" that have to be learned (these are the "weighted" **W** between layers)

Convolutional Neural Network (CNN)

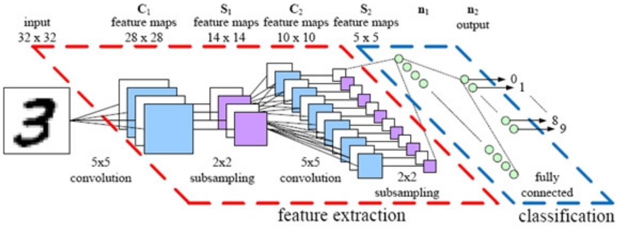
- Applies 2 operations, convolution and pooling (subsampling), repeatedly on the input data



- Convolution: Extract "local" properties of the signal. Uses a set of "filters" that have to be learned (these are the "weighted" $\mathbf{W}$ between layers)
- Pooling: Downsamples the outputs to reduce the size of representation

Convolutional Neural Network (CNN)

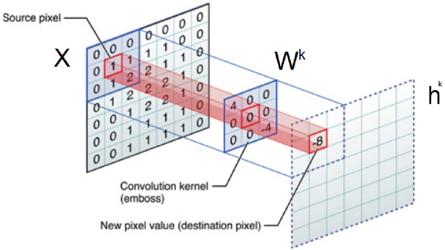
- Applies 2 operations, convolution and pooling (subsampling), repeatedly on the input data



- Convolution: Extract "local" properties of the signal. Uses a set of "filters" that have to be learned (these are the "weighted" **W** between layers)
- Pooling: Downsamples the outputs to reduce the size of representation
- Note: A nonlinearity is also introduced after the convolution layer

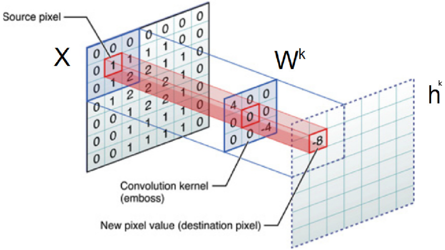
Convolution

- An operation that captures local (e.g., spatial) properties of a signal



Convolution

- An operation that captures local (e.g., spatial) properties of a signal



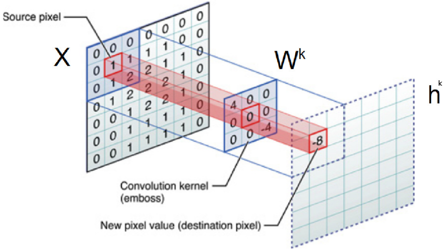
- Mathematically, the operation is defined as

$$h_{ij}^k = g((W^k * \mathbf{X})_{ij} + b_k)$$

where W^k is a filter, $*$ is the convolution operator, and g is a nonlinearity

Convolution

- An operation that captures local (e.g., spatial) properties of a signal



- Mathematically, the operation is defined as

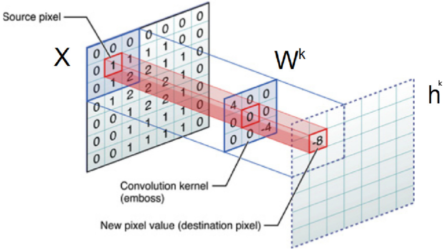
$$h_{ij}^k = g((W^k * \mathbf{X})_{ij} + b_k)$$

where W^k is a filter, $*$ is the convolution operator, and g is a nonlinearity

- Usually several filters $\{W^k\}_{k=1}^K$ are applied (each will produce a separate "feature map")

Convolution

- An operation that captures local (e.g., spatial) properties of a signal



- Mathematically, the operation is defined as

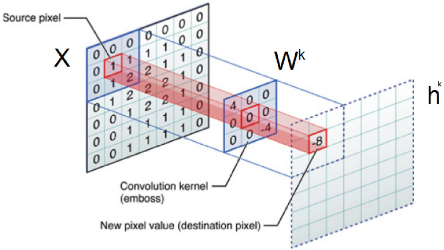
$$h_{ij}^k = g((W^k * \mathbf{X})_{ij} + b_k)$$

where W^k is a filter, $*$ is the convolution operator, and g is a nonlinearity

- Usually several filters $\{W^k\}_{k=1}^K$ are applied (each will produce a separate "feature map"). These filters have to be learned (these are the weights of the NN)

Convolution

- An operation that captures local (e.g., spatial) properties of a signal



- Mathematically, the operation is defined as

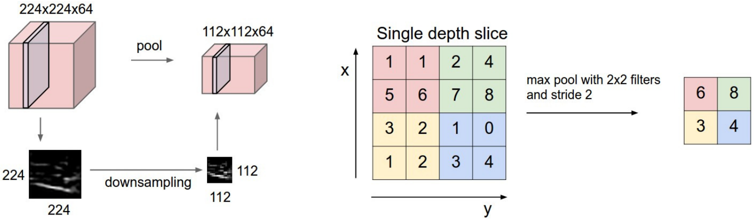
$$h_{ij}^k = g((W^k * \mathbf{X})_{ij} + b_k)$$

where W^k is a filter, $*$ is the convolution operator, and g is a nonlinearity

- Usually several filters $\{W^k\}_{k=1}^K$ are applied (each will produce a separate "feature map"). These filters have to be learned (these are the weights of the NN)

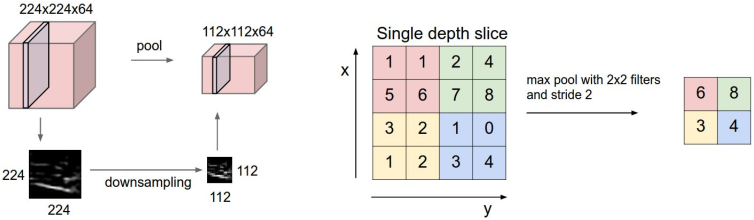
Pooling/Downsampling

- Used to "downsample" the representation-size after convolution step.



Pooling/Downsampling

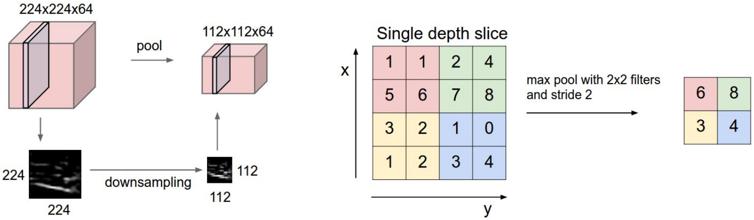
- Used to "downsample" the representation-size after convolution step.



- Also ensures robustness against minor rotations, shifts, corruptions in the image

Pooling/Downsampling

- Used to "downsample" the representation-size after convolution step.



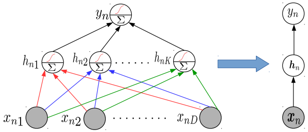
- Also ensures robustness against minor rotations, shifts, corruptions in the image
- Popular approaches: Max-pooling, averaging pooling, etc

Strides

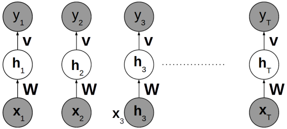
- Stride defines the number of nodes a filter moves between two consecutive convolution operations
- Likewise, we have a stride to define the same when applying pooling

Modeling Sequential Data

- FFNN for a single observation looks like this (denoting all hidden units as $\boldsymbol{h}_n$)

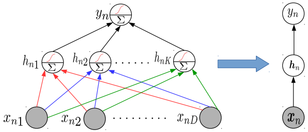


- FFNN can't take into account the structure in sequential data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$, e.g., it would look like

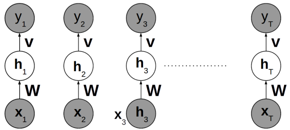


Modeling Sequential Data

- FFNN for a single observation looks like this (denoting all hidden units as $\boldsymbol{h}_n$)



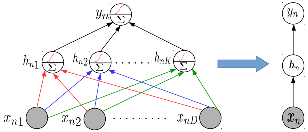
- FFNN can't take into account the structure in sequential data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$, e.g., it would look like



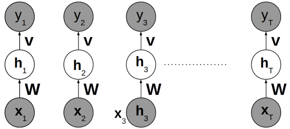
- For such sequential data, we want dependencies between $\boldsymbol{h}_t$'s of different observations

Modeling Sequential Data

- FFNN for a single observation looks like this (denoting all hidden units as $\boldsymbol{h}_n$)



- FFNN can't take into account the structure in sequential data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$, e.g., it would look like



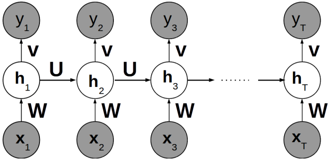
- For such sequential data, we want dependencies between $\boldsymbol{h}_t$'s of different observations
- Desirable when modeling sentence/paragraph/document, video (sequence of frames), etc.

Recurrent Neural Nets (RNN)

- A simple neural network for sequential data

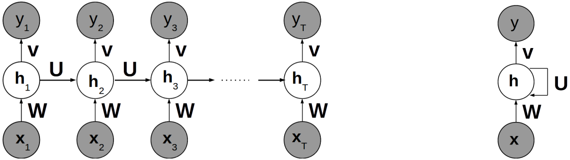
Recurrent Neural Nets (RNN)

- A simple neural network for sequential data
- Hidden state at each step depends on the hidden state of the previous



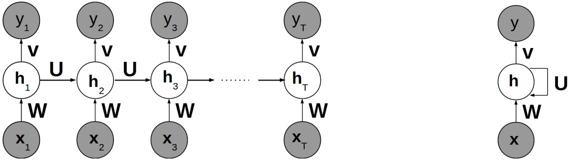
Recurrent Neural Nets (RNN)

- A simple neural network for sequential data
- Hidden state at each step depends on the hidden state of the previous



Recurrent Neural Nets (RNN)

- A simple neural network for sequential data
- Hidden state at each step depends on the hidden state of the previous



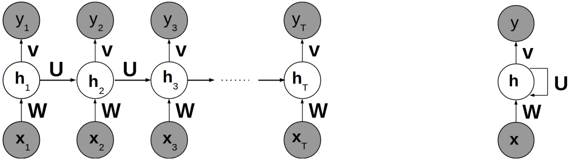
- Each hidden state is typically defined as

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

where $\mathbf{U}$ is a $K \times K$ transition matrix and f is some nonlin. fn. (e.g., tanh)

Recurrent Neural Nets (RNN)

- A simple neural network for sequential data
- Hidden state at each step depends on the hidden state of the previous



- Each hidden state is typically defined as

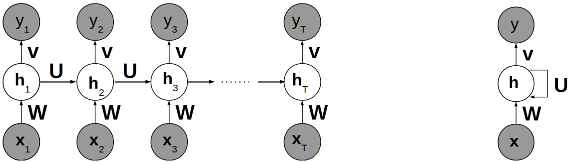
$$\boldsymbol{h}_t = f(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

  where $\mathbf{U}$ is a $K \times K$ transition matrix and f is some nonlin. fn. (e.g., tanh)

- Now $\boldsymbol{h}_t$ acts as a "memory". Helps us remember what happened up to step t

Recurrent Neural Nets (RNN)

- A simple neural network for sequential data
- Hidden state at each step depends on the hidden state of the previous



- Each hidden state is typically defined as

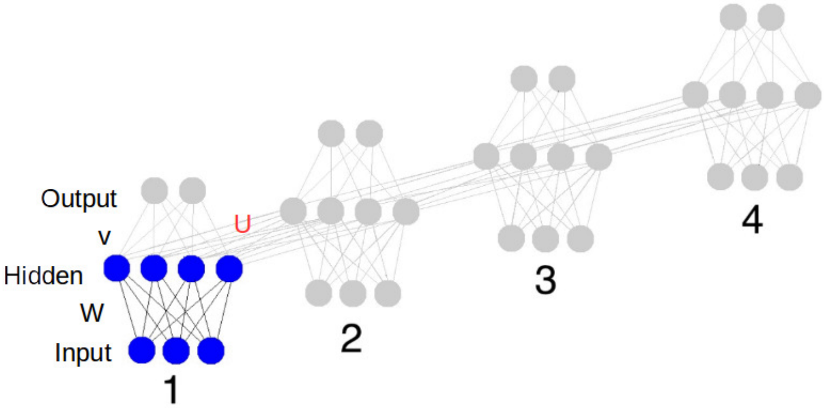
$$\boldsymbol{h}_t = f(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

  where $\mathbf{U}$ is a $K \times K$ transition matrix and f is some nonlin. fn. (e.g., tanh)

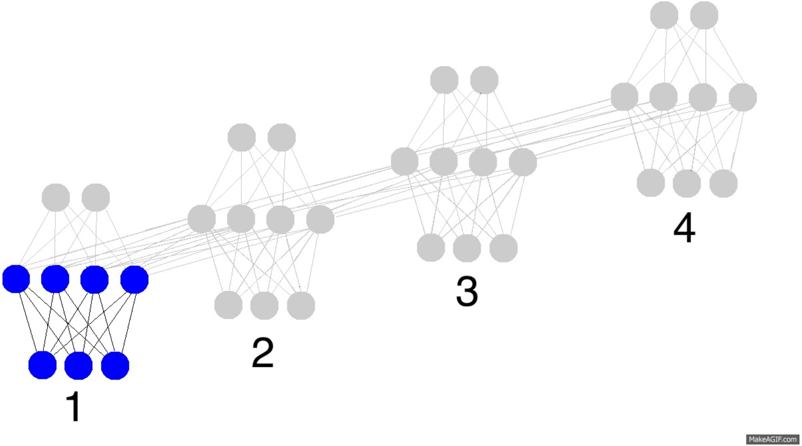
- Now $\boldsymbol{h}_t$ acts as a "memory". Helps us remember what happened up to step t
- RNNs can also be extended to have more than one hidden layer

Recurrent Neural Nets (RNN)

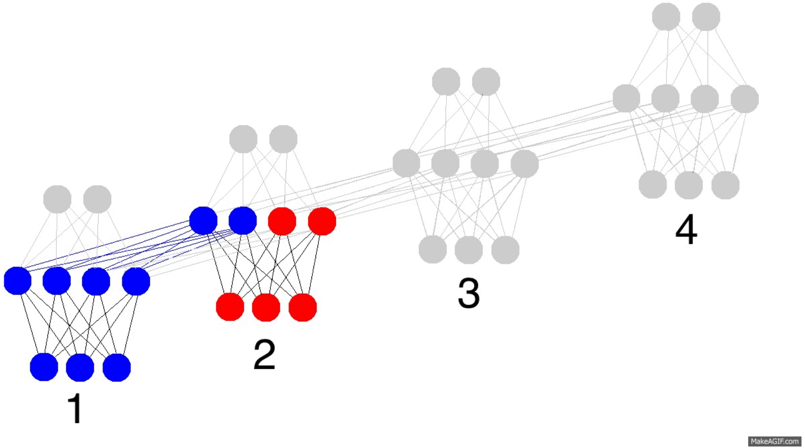
- A more "micro" view of RNN (the transition matrix **U** connects the hidden states across observations, propagating information along the sequence)



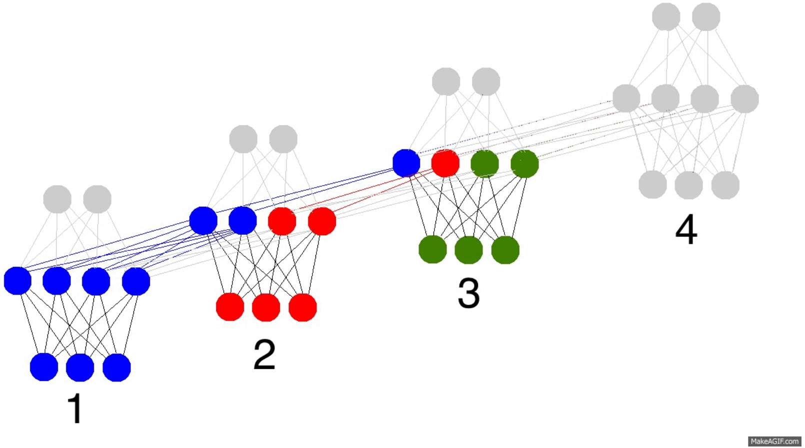
RNN in Action..



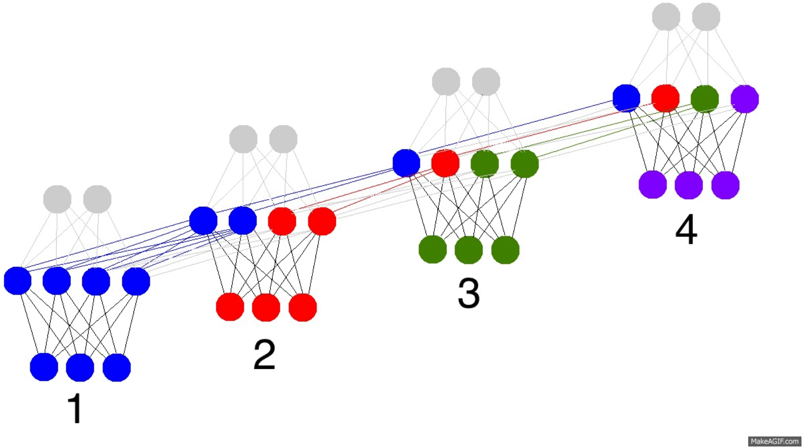
RNN in Action..



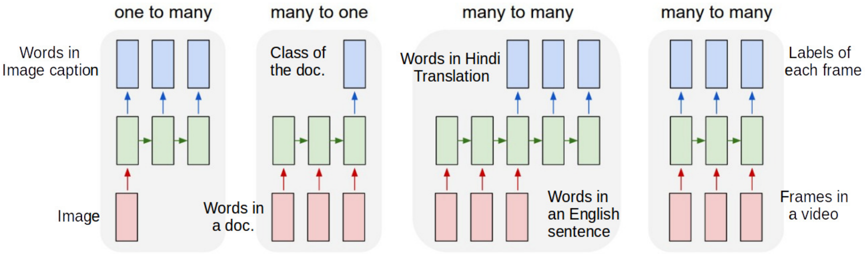
RNN in Action..



RNN in Action..

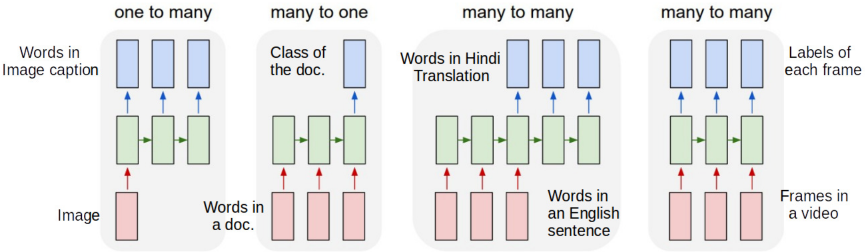


RNN: Applications



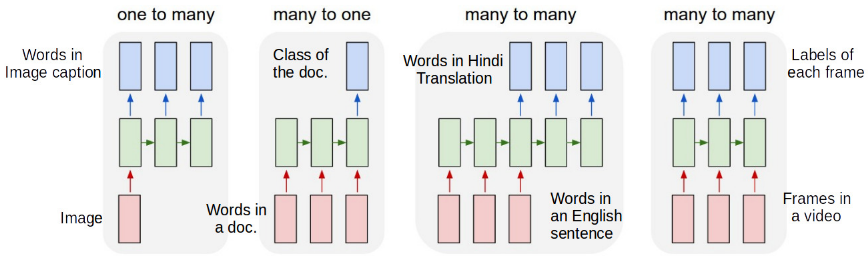
- RNNs are widely applicable and are also very flexible.

RNN: Applications



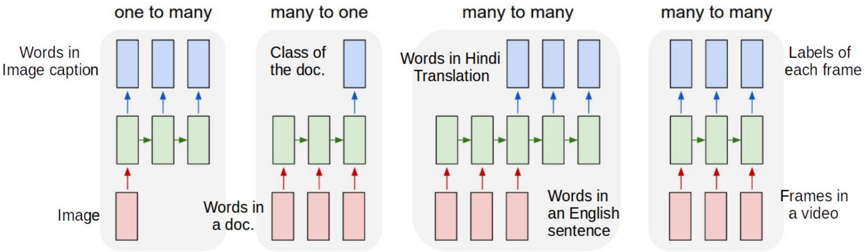
- RNNs are widely applicable and are also very flexible. E.g.,

RNN: Applications



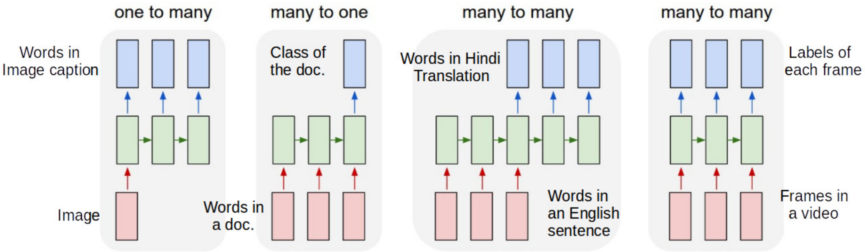
- RNNs are widely applicable and are also very flexible. E.g.,
  - Input, output, or both, can be sequences (possibly of different lengths)

RNN: Applications



- RNNs are widely applicable and are also very flexible. E.g.,
  - Input, output, or both, can be sequences (possibly of different lengths)
  - Different inputs (and different outputs) need not be of the same length

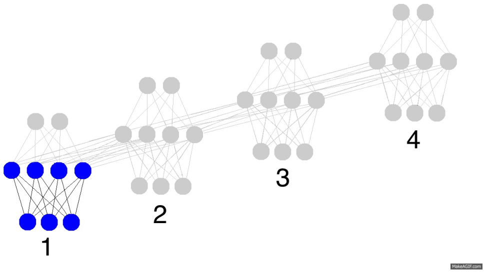
RNN: Applications



- RNNs are widely applicable and are also very flexible. E.g.,
  - Input, output, or both, can be sequences (possibly of different lengths)
  - Different inputs (and different outputs) need not be of the same length
  - Regardless of the length of the input sequence, RNN will learn a fixed size embedding for the input sequence

Training RNN

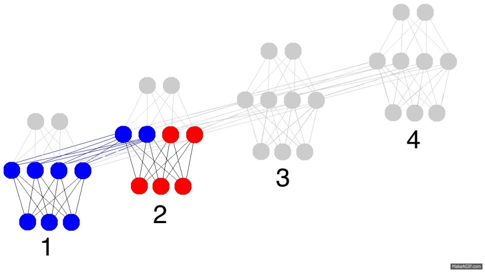
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

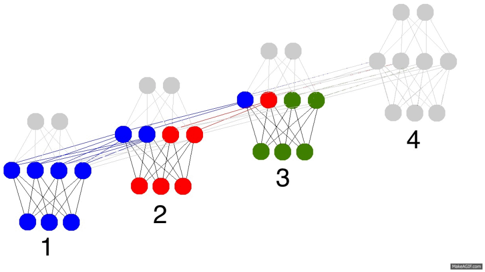
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

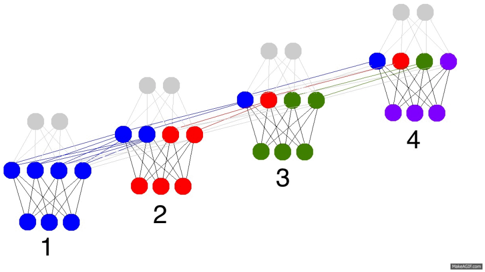
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

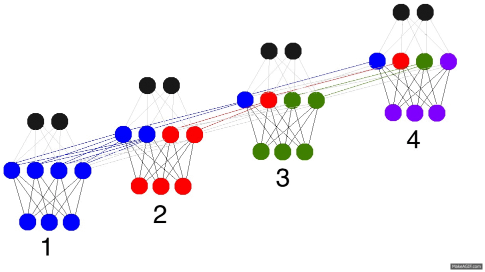
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

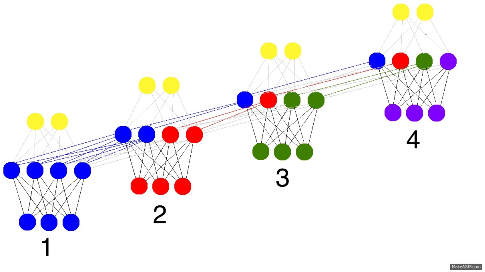
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

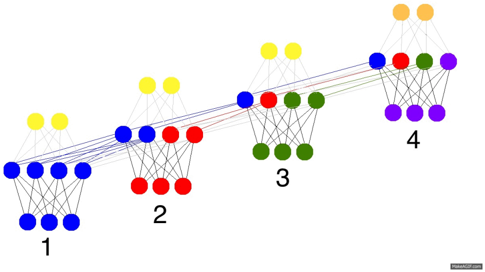
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

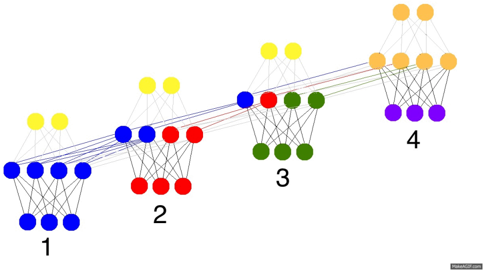
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

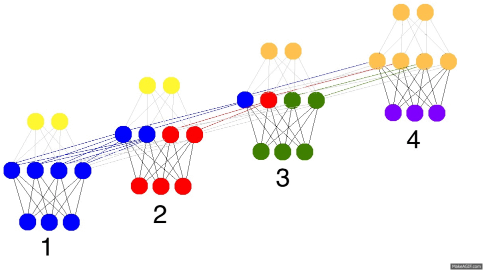
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

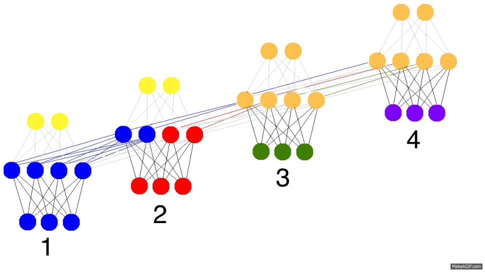
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

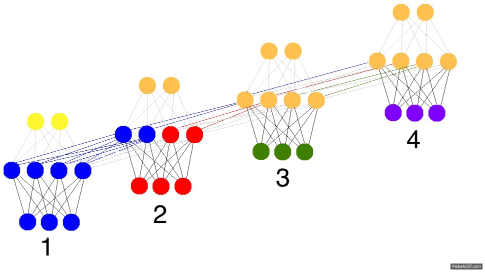
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

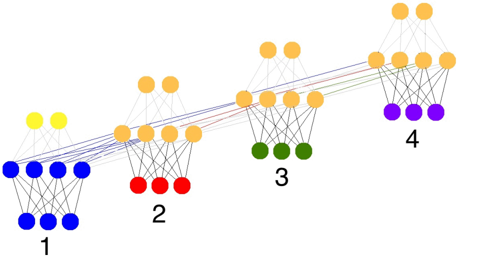
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

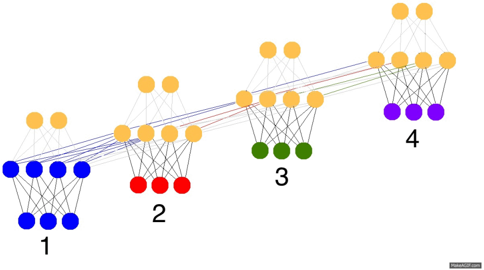
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

Training RNN

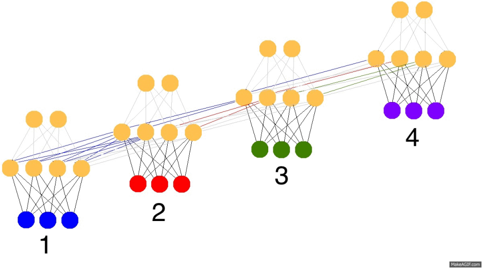
- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



- Black: Prediction, Yellow: Error, Orange: Gradients

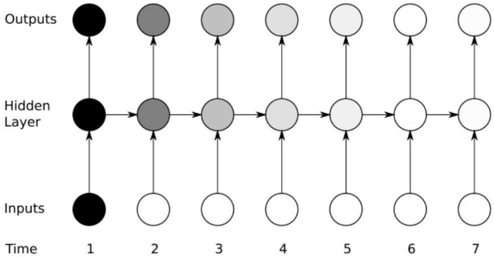
Training RNN

- Trained using Backpropagation Through Time (forward propagate from step 1 to end, and then backward propagate from end to step 1)
- Think of the time-dimension as another hidden layer and then it is just like standard backpropagation for feedforward neural nets



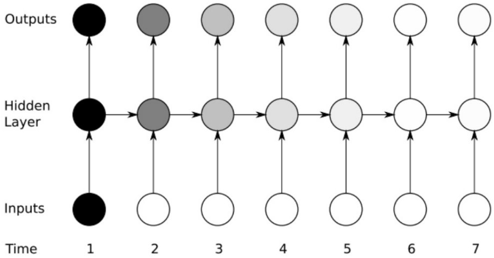
- Black: Prediction, Yellow: Error, Orange: Gradients

RNN Limitation



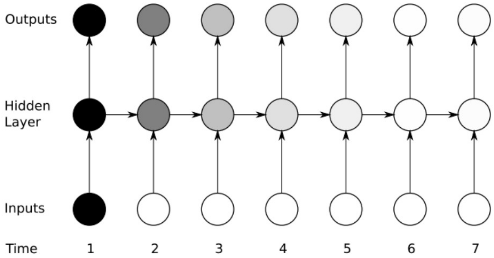
- Sensitivity of hidden states and outputs on a given input becomes weaker as we move away from it along the sequence (weak memory)

RNN Limitation



- Sensitivity of hidden states and outputs on a given input becomes weaker as we move away from it along the sequence (weak memory)
- New inputs "overwrite" the activations of previous hidden states

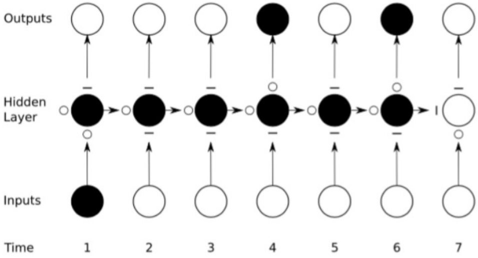
RNN Limitation



- Sensitivity of hidden states and outputs on a given input becomes weaker as we move away from it along the sequence (weak memory)
- New inputs "overwrite" the activations of previous hidden states
- Repeated multiplications can cause the gradients to vanish or explode

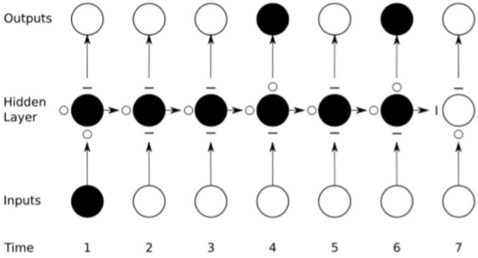
Capturing Long-Range Dependencies

- Idea: Augment the hidden states with gates (with parameters to be learned)
- These gates can help us remember and forget information "selectively"



Capturing Long-Range Dependencies

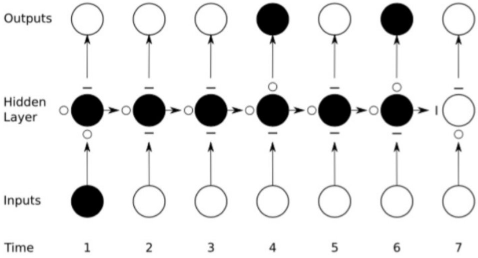
- Idea: Augment the hidden states with gates (with parameters to be learned)
- These gates can help us remember and forget information "selectively"



- The hidden states have 3 type of gates
  - Input (bottom), Forget (left), Output (top)

Capturing Long-Range Dependencies

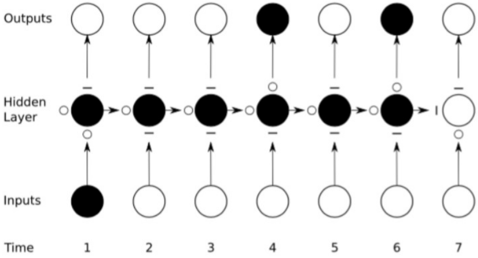
- Idea: Augment the hidden states with gates (with parameters to be learned)
- These gates can help us remember and forget information "selectively"



- The hidden states have 3 type of gates
  - Input (bottom), Forget (left), Output (top)
- Open gate denoted by 'o', closed gate denoted by '-'

Capturing Long-Range Dependencies

- Idea: Augment the hidden states with gates (with parameters to be learned)
- These gates can help us remember and forget information "selectively"



- The hidden states have 3 type of gates
  - Input (bottom), Forget (left), Output (top)
- Open gate denoted by 'o', closed gate denoted by '-'
- LSTM (Hochreiter and Schmidhuber, mid-90s): **Long Short-Term Memory** is one such idea

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a “context” C_t and computes hidden states as

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\hat{C}_t = \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) \quad \text{("local" context, only up to immediately preceding state)}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\hat{C}_t = \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) \quad \text{("local" context, only up to immediately preceding state)}$$

$$i_t = \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) \quad \text{(how much to take in the local context)}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) && \text{("local" context, only up to immediately preceding state)} \\
{\color{blue}i_t} &= \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) && \text{(how much to take {\color{blue}in} the local context)} \\
{\color{red}f_t} &= \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) && \text{(how much to {\color{red}forget} the previous context)}
\end{aligned}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\hat{C}_t = \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) \quad \text{("local" context, only up to immediately preceding state)}$$

$$i_t = \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) \quad \text{(how much to take in the local context)}$$

$$f_t = \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) \quad \text{(how much to forget the previous context)}$$

$$o_t = \sigma(\mathbf{W}^o\boldsymbol{x}_t + \mathbf{U}^o\boldsymbol{h}_{t-1}) \quad \text{(how much to output)}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) && \text{("local" context, only up to immediately preceding state)} \\
i_t &= \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) && \text{(how much to take in the local context)} \\
f_t &= \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) && \text{(how much to forget the previous context)} \\
o_t &= \sigma(\mathbf{W}^o\boldsymbol{x}_t + \mathbf{U}^o\boldsymbol{h}_{t-1}) && \text{(how much to output)} \\
C_t &= C_{t-1} \odot f_t + \hat{C}_t \odot i_t && \text{(a modulated additive update for context)}
\end{aligned}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) && \text{("local" context, only up to immediately preceding state)}\\
i_t &= \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) && \text{(how much to take in the local context)}\\
f_t &= \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) && \text{(how much to forget the previous context)}\\
o_t &= \sigma(\mathbf{W}^o\boldsymbol{x}_t + \mathbf{U}^o\boldsymbol{h}_{t-1}) && \text{(how much to output)}\\
C_t &= C_{t-1} \odot f_t + \hat{C}_t \odot i_t && \text{(a modulated additive update for context)}\\
h_t &= \tanh(C_t) \odot o_t && \text{(transform context into state and selectively output)}
\end{aligned}$$

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) && \text{("local" context, only up to immediately preceding state)}\\
i_t &= \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) && \text{(how much to take in the local context)}\\
f_t &= \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) && \text{(how much to forget the previous context)}\\
o_t &= \sigma(\mathbf{W}^o\boldsymbol{x}_t + \mathbf{U}^o\boldsymbol{h}_{t-1}) && \text{(how much to output)}\\
C_t &= C_{t-1} \odot f_t + \hat{C}_t \odot i_t && \text{(a modulated additive update for context)}\\
h_t &= \tanh(C_t) \odot o_t && \text{(transform context into state and selectively output)}
\end{aligned}$$

- Note: $\odot$ represents elementwise vector product. Also, state updates now additive, not multiplicative. Training using backpropagation through time.

Long Short-Term Memory (LSTM)

- Essentially an RNN, except that the hidden states are computed differently
- Recall that RNN computes the hidden states as

$$\boldsymbol{h}_t = \tanh(\mathbf{W}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1})$$

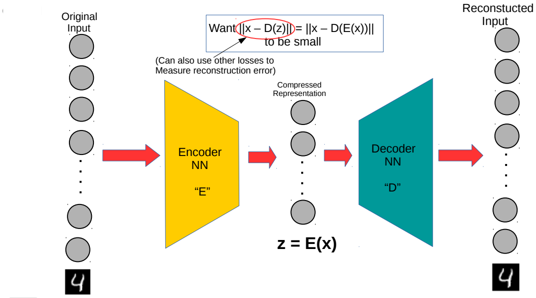
- For RNN: State update is multiplicative (weak memory and gradient issues)
- In contrast, LSTM maintains a "context" C_t and computes hidden states as

$$\begin{aligned}
\hat{C}_t &= \tanh(\mathbf{W}^c\boldsymbol{x}_t + \mathbf{U}^c\boldsymbol{h}_{t-1}) && \text{(“local” context, only up to immediately preceding state)}\\
i_t &= \sigma(\mathbf{W}^i\boldsymbol{x}_t + \mathbf{U}^i\boldsymbol{h}_{t-1}) && \text{(how much to take in the local context)}\\
f_t &= \sigma(\mathbf{W}^f\boldsymbol{x}_t + \mathbf{U}^f\boldsymbol{h}_{t-1}) && \text{(how much to forget the previous context)}\\
o_t &= \sigma(\mathbf{W}^o\boldsymbol{x}_t + \mathbf{U}^o\boldsymbol{h}_{t-1}) && \text{(how much to output)}\\
C_t &= C_{t-1} \odot f_t + \hat{C}_t \odot i_t && \text{(a modulated additive update for context)}\\
h_t &= \tanh(C_t) \odot o_t && \text{(transform context into state and selectively output)}
\end{aligned}$$

- Note: $\odot$ represents elementwise vector product. Also, state updates now additive, not multiplicative. Training using backpropagation through time.
- Many variants of LSTM exists, e.g., using C_{t-1} in local computations, Gated Recurrent Units (GRU), etc. Mostly minor variations of basic LSTM above

Deep Neural Networks for Unsupervised Learning

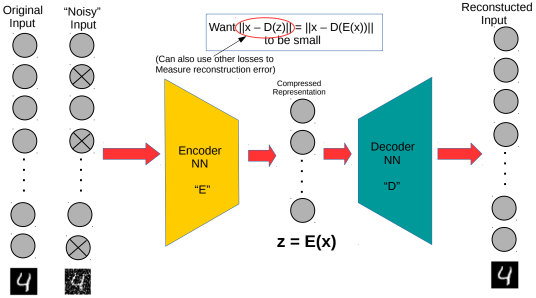
- Auto-encoder (AE) is a popular deep neural network unsupervised feature learning



- If size $\boldsymbol{z}$ is $K < D$, auto-encoders can be used for dimensionality reduction too
- For linear encoder/decodder with $E(\boldsymbol{x}) = \mathbf{W}^\top \boldsymbol{x}$, $D(\boldsymbol{z}) = \mathbf{W}\boldsymbol{z}$ and squared loss, AE is akin to PCA

Deep Neural Networks for Unsupervised Learning

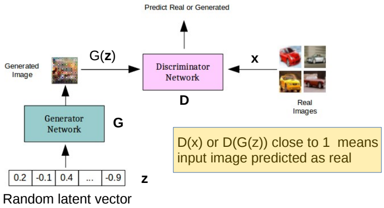
- Denoising auto-encoders: Inject noise in the inputs before passing to to encoder



- Many ways to introduct "noise": Inject zero-mean Gaussian noise, "zero-out" some features, etc
- Especially useful when $K > D$ ($\boldsymbol{z}$ to be a copy of $\boldsymbol{x}$ with $K - D$ zeros) - overcomplete autocoders

Generative Adversarial Network

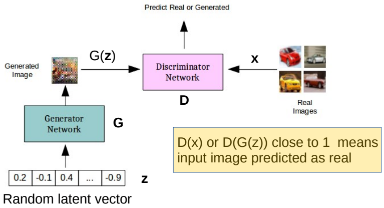
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

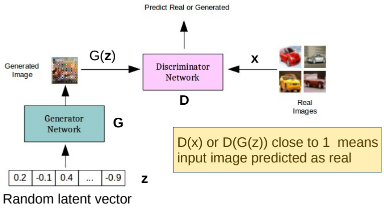
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

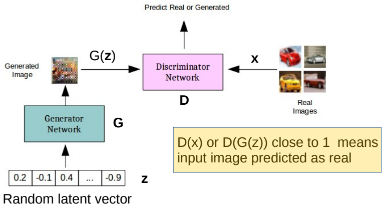
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

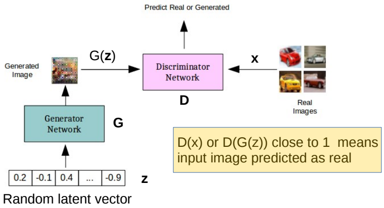
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data
- Generator transforms a random $\boldsymbol{z}$ to produce fake data



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

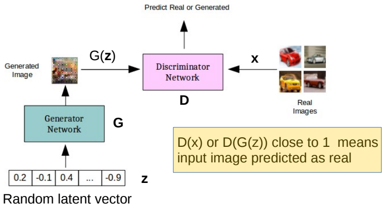
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data
- Generator transforms a random $\boldsymbol{z}$ to produce fake data
- Discriminator's Goal: Make $D(x) \rightarrow 1$, $D(G(z)) \rightarrow 0$



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

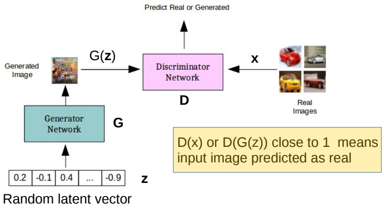
- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data
- Generator transforms a random $\boldsymbol{z}$ to produce fake data
- Discriminator's Goal: Make $D(x) \to 1$, $D(G(z)) \to 0$
- Generator's Goal: Make $D(G(z)) \to 1$ (fool discr.)



$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Generative Adversarial Network

- A model that can learn to generate highly real looking data (Goodfellow et al, 2014)
- A game between a Generator and a Discriminator
- Both are modeled by deep neural networks
- Discriminator: A classifier to predict real vs fake data
- Generator transforms a random $\boldsymbol{z}$ to produce fake data
- Discriminator's Goal: Make $D(x) \rightarrow 1$, $D(G(z)) \rightarrow 0$
- Generator's Goal: Make $D(G(z)) \rightarrow 1$ (fool discr.)
- At the game's equilibrium, the generator starts producing data from the true data distribution $p_{data}(x)$



$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Some Other Advances..

- Deep Probabilistic Models: The linear probabilistic models we've seen can be "deep-ified"
- Basically, just require changing the linear part by a (deep) NN

Some Other Advances..

- Deep Probabilistic Models: The linear probabilistic models we've seen can be "deep-ified"
- Basically, just require changing the linear part by a (deep) NN , e.g.,
    - Deep probabilistic model for regression/classification

$$
\begin{aligned}
y_n &\sim \mathcal{N}(y_n|\text{NN}(\boldsymbol{x}_n), \beta^{-1}) \\
y_n &\sim \text{Bernoulli}(y_n|\sigma(\text{NN}(\boldsymbol{x}_n)))
\end{aligned}
$$

Some Other Advances..

- Deep Probabilistic Models: The linear probabilistic models we've seen can be "deep-ified"
- Basically, just require changing the linear part by a (deep) NN , e.g.,
  - Deep probabilistic model for regression/classification

$$
\begin{aligned}
y_n &\sim \mathcal{N}(y_n|\text{NN}(\boldsymbol{x}_n), \beta^{-1}) \\
y_n &\sim \text{Bernoulli}(y_n|\sigma(\text{NN}(\boldsymbol{x}_n)))
\end{aligned}
$$

  - Deep probabilistic PPCA; a.k.a. variational autoencoder (VAE)

$$
\begin{aligned}
\boldsymbol{z}_n &\sim \mathcal{N}(0, \mathbf{I}_K) \\
\boldsymbol{x}_n &\sim \mathcal{N}(\boldsymbol{x}_n|\text{NN}_\mu(\boldsymbol{z}_n), \text{NN}_{\sigma^2}(\boldsymbol{z}_n))
\end{aligned}
$$

Some Other Advances..

- Deep Probabilistic Models: The linear probabilistic models we've seen can be "deep-ified"
- Basically, just require changing the linear part by a (deep) NN , e.g.,
  - Deep probabilistic model for regression/classification

$$
\begin{aligned}
y_n &\sim \mathcal{N}(y_n|\text{NN}(\boldsymbol{x}_n), \beta^{-1}) \\
y_n &\sim \text{Bernoulli}(y_n|\sigma(\text{NN}(\boldsymbol{x}_n)))
\end{aligned}
$$

  - Deep probabilistic PPCA; a.k.a. variational autoencoder (VAE)

$$
\begin{aligned}
\boldsymbol{z}_n &\sim \mathcal{N}(0, \mathbf{I}_K) \\
\boldsymbol{x}_n &\sim \mathcal{N}(\boldsymbol{x}_n|\text{NN}_\mu(\boldsymbol{z}_n), \text{NN}_{\sigma^2}(\boldsymbol{z}_n))
\end{aligned}
$$

- Can do MAP estimation of the NN parameters or even infer full posterior (Bayesian Deep Learning)

Some Concluding Comments

- Deep Learning is extremely popular and topical

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics
- Deep Learning is not the necessarily the best way to do ML :-)

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics
- Deep Learning is not the necessarily the best way to do ML :-)
- Many non-deep learning methods can often perform comparably (sometimes even better)..

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics
- Deep Learning is not the necessarily the best way to do ML :-)
- Many non-deep learning methods can often perform comparably (sometimes even better)..
    - Decision trees, kernel methods, mixture-of-experts, and others..

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics
- Deep Learning is not the necessarily the best way to do ML :-)
- Many non-deep learning methods can often perform comparably (sometimes even better)..
    - Decision trees, kernel methods, mixture-of-experts, and others..
- Therefore don't abandon the other methods we have learned in the course :-)

Some Concluding Comments

- Deep Learning is extremely popular and topical
- Impressive success in many areas such as vision, NLP, robotics
- Deep Learning is not the necessarily the best way to do ML :-)
- Many non-deep learning methods can often perform comparably (sometimes even better)..
  - Decision trees, kernel methods, mixture-of-experts, and others..
- Therefore don't abandon the other methods we have learned in the course :-)
- We are yet to see other non-deep learning methods that are very valuable