Deep Generative Models

Affective Computing

Taher Ahmadi

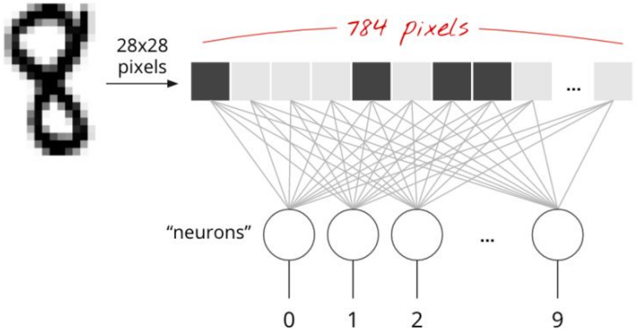
Summer 2020

What's a NN?

- Let's start with an example:
- Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images.

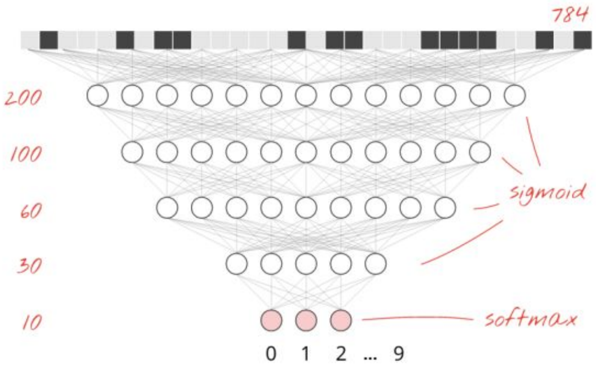
One-Layer Network For Classifying MNIST

- Let's make a one-layer neural network for classifying digits.
- Each neuron in a neural network:
  - Does a weighted sum of all of its inputs
  - Adds a bias
  - Feeds the result through some non-linear activation function, e.g., softmax.



Vanilla Deep Neural Network

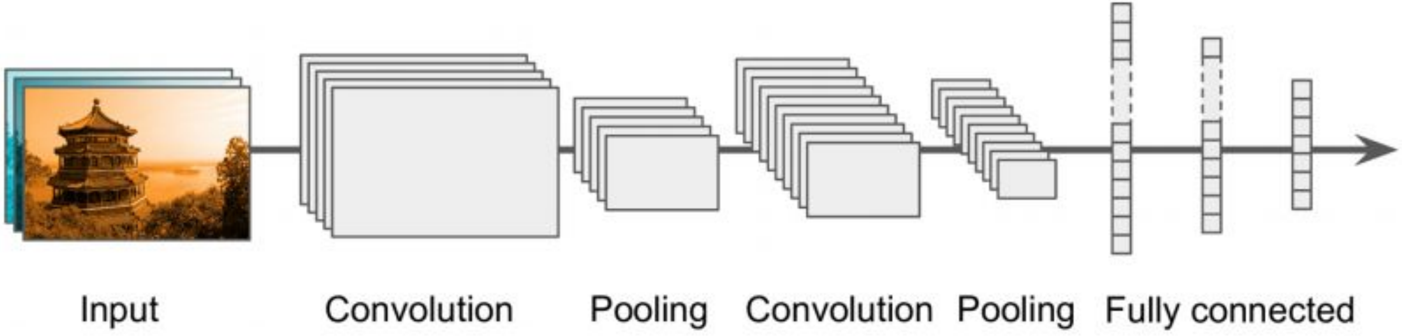
- Add more layers to improve the accuracy.
- On intermediate layers we will use the the sigmoid activation function.
- We keep softmax as the activation function on the last layer.



[https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd]

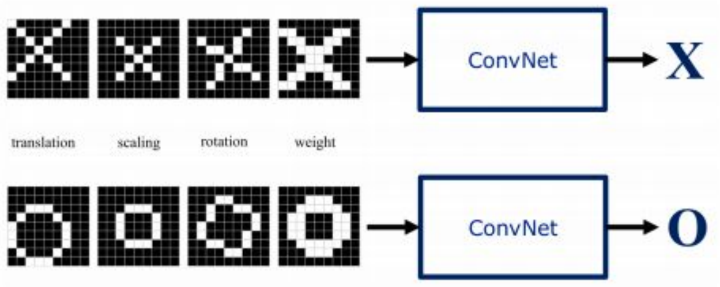
Convolutional Neural Networks

- Convolutional layers: apply a specified number of convolution filters to the image.
- Pooling layers: downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time.
- Dense layers: a fully connected layer that performs classification on the features extracted by the convolutional layers and downsampled by the pooling layers.

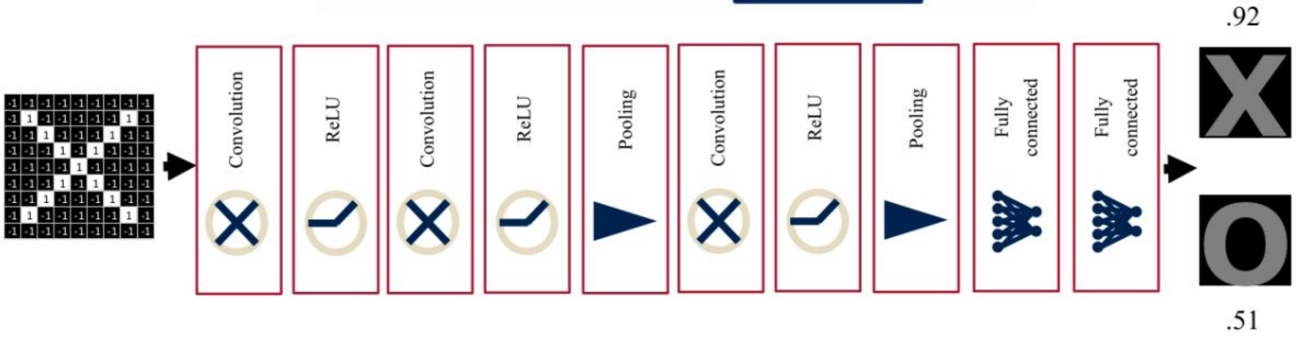


CNN for X/O

A two-dimensional
array of pixels
ConvNet
X or O



ConvNet
X
translation
scaling
rotation
weight
ConvNet
O



Convolution
ReLU
Convolution
ReLU
Pooling
Convolution
ReLU
Pooling
Fully connected
Fully connected
.92
.51

Neural Networks in Keras

Keras

- Keras is a high-level API to build and train deep learning models.
- To get started, import tf.keras to your python code.

```python
import tensorflow as tf
from tensorflow.keras import layers
```

Keras Layers

- In Keras, you assemble layers tf.keras.layers to build models.
- A model is (usually) a graph of layers.
- There are many types of layers, e.g., Dense, Conv2D, RNN, ...

Keras Layers

- Common constructor parameters:
  - activation: the activation function for the layer.
  - kernel initializer and bias initializer: the initialization schemes of the layer’s weights.
  - kernel regularizer and bias regularizer: the regularization schemes of the layer’s weights, e.g., L1 or L2.

```
layers.Dense(64, activation=tf.sigmoid, kernel_regularizer=tf.keras.regularizers.l1(0.01),
bias_initializer=tf.keras.initializers.constant(2.0))
```

Keras Models

- There are two ways to build Keras models: sequential and functional.
- The sequential API allows you to create models layer-by-layer.
- The functional API allows you to create models that have a lot more flexibility.
  - You can define models where layers connect to more than just their previous and next layers.

Keras Models - Sequential Models

- You can use tf.keras.Sequential to build a sequential model.

```
from tensorflow.keras import layers

model = tf.keras.Sequential()

model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
```

Keras Models - Functional Models

- You can use tf.keras.Model to build a functional model.

```
from tensorflow.keras import layers

inputs = tf.keras.Input(shape=(32,32))
x = layers.Dense(64, activation="relu")(inputs)
x = layers.Dense(64, activation="relu")(x)
predictions = layers.Dense(10, activation="softmax")(x)

model = tf.keras.Model(inputs=inputs, outputs=predictions)
```

Training Keras Models

- Call the compile method to configure the learning process.
- tf.keras.Model.compile takes three important arguments.
    - optimizer: specifies the training procedure.
    - loss: the cost function to minimize during optimization, e.g., mean squared error (mse), categorical_crossentropy, and binary_crossentropy.
    - metrics: used to monitor training.
- Call the fit() method to fit the model the training data.

```
model.compile(optimizer=tf.train.GradientDescentOptimizer(0.001), loss="mse", metrics=["mae"])

model.fit(training_data, training_labels, epochs=10, batch_size=32)
```

Evaluate and Predict

- tf.keras.Model.evaluate: evaluate the cost and metrics for the data provided.
- tf.keras.Model.predict: predict the output of the last layer for the data provided.

```
model.evaluate(test_data, test_labels, batch_size=32)

model.predict(test_data, batch_size=32)
```

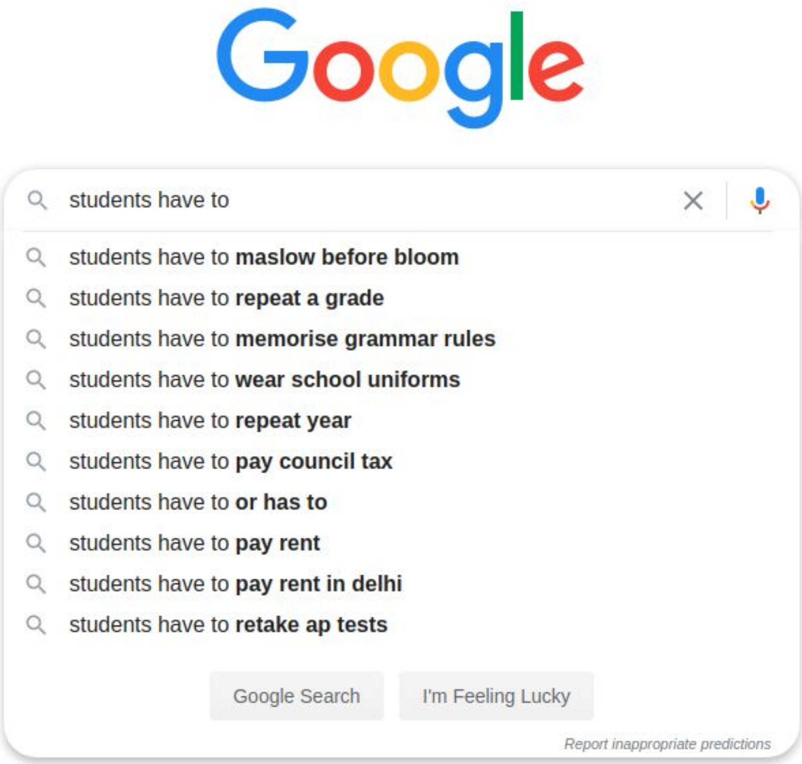
Feedforward Network in Keras

- Building and Training a two layer sequential model in Keras.

```
n_neurons_h = 4
n_neurons_out = 3
n_epochs = 100
learning_rate = 0.1

model = tf.keras.Sequential()

model.add(layers.Dense(n_neurons_h, activation="sigmoid"))
model.add(layers.Dense(n_neurons_out, activation="sigmoid"))

model.compile(optimizer=tf.train.GradientDescentOptimizer(learning_rate.001),
loss="binary_crossentropy", metrics=["accuracy"])
model.fit(training_X, training_y, epochs=n_epochs)
```

RNNs

An example...



- Language modeling is the task of predicting what word comes next

n-gram Language Models

- The students have to ...
- How to learn a Language Model?
- Learn a n-gram Language Model!
- A n-gram is a chunk of n consecutive words.
    - Unigrams: "the", "students", "have", "to"
    - Bigrams: "the students", "students have", "have to"
    - Trigrams: "the students have", "students have to"
    - 4-grams: "the students have to"
    - ...
- Collect statistics about how frequent different n-grams are, and use these to predict next word.

p(w_j | students have to) = students have to w_j / students have to

Problems with n-gram Language Models

- What if "students have to w_j" never occurred in data?
  - Then w_j has probability 0!
- What if "students have to" never occurred in data?
  - Then we can’t calculate probability for any w_j!
- Increasing n makes sparsity problems worse.
  - Typically we can’t have n bigger than 5.
- Increasing n makes model size huge.

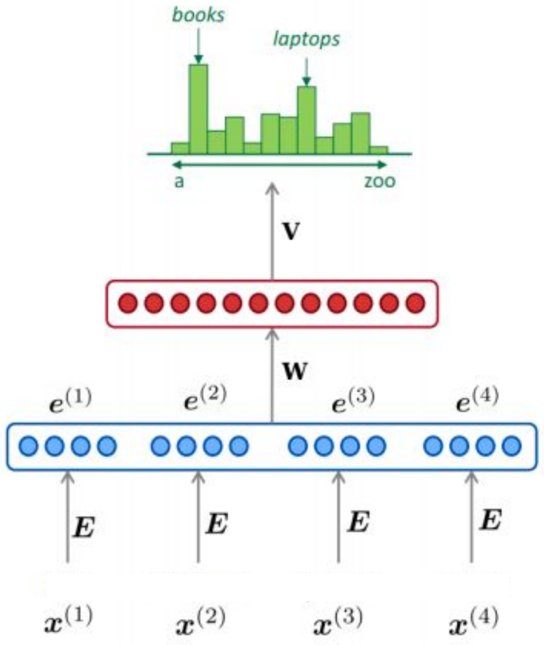
Can we use a Neural Model?

Recall the Language Modeling task:

- Input: sequence of words $x^{(1)}, x^{(2)}, \cdots, x^{(t)}$
- Output: probability dist of the next word $p(x^{(t+1)} = w_j \mid x^{(t)}, \cdots, x^{(1)})$

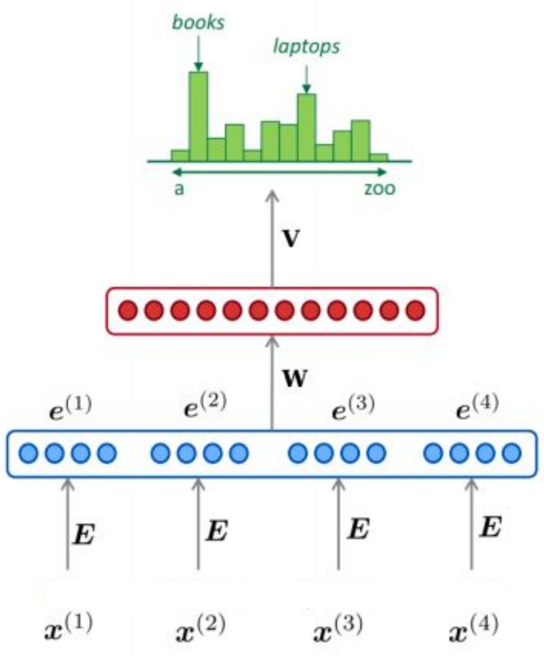
- MLP model:

  Input: words $\mathrm{x}^{(1)}, \mathrm{x}^{(2)}, \mathrm{x}^{(3)}, \mathrm{x}^{(4)}$
  Input layer: one-hot vectors $\mathbf{e}^{(1)}, \mathbf{e}^{(2)}, \mathbf{e}^{(3)}, \mathbf{e}^{(4)}$
  Hidden layer: $\mathbf{h} = \mathtt{f}(\mathbf{w}^\intercal \mathbf{e})$, $\mathtt{f}$ is an activation function.
  Output: $\hat{\mathbf{y}} = \mathtt{softmax}(\mathbf{v}^\intercal \mathbf{h})$



Can we use a Neural Model?

- Improvements over n-gram LM:
  - No sparsity problem
  - Model size is O(n) not O(exp(n))
- Remaining problems:
  - It is fixed 4 in our example, which is small
  - We need a neural architecture that can process any length input

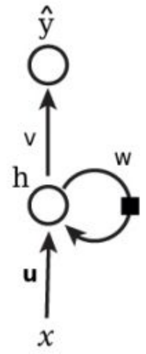


Recurrent Neural Networks (RNN's)

- The idea behind Recurrent neural networks (RNN) is to make use of sequential data.
  - Until here, we assume that all inputs (and outputs) are independent of each other.
  - It is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).
- They can analyze time series data and predict the future.
- They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs.

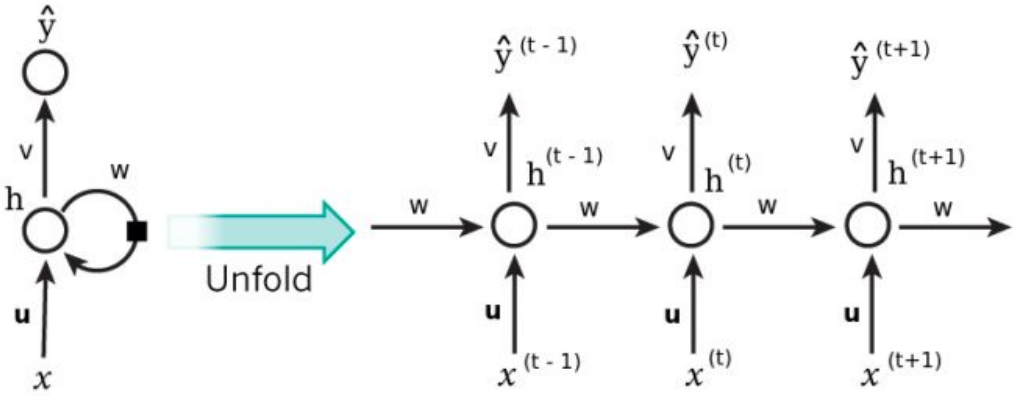
Recurrent Neural Networks

- Neurons in an RNN have connections pointing backward.
- RNNs have memory, which captures information about what has been calculated so far.



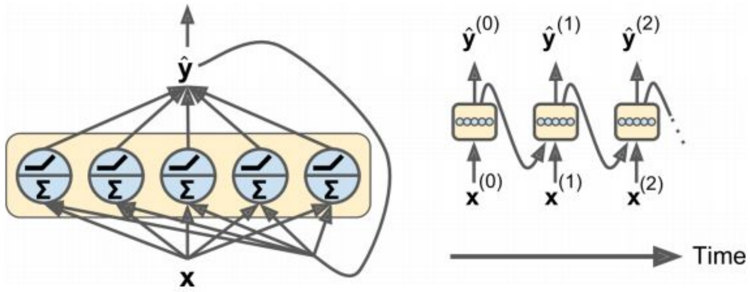
Recurrent Neural Networks

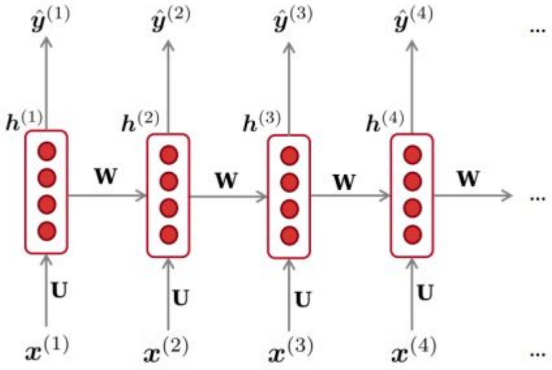
- Unfolding the network: represent a network against the time axis.
  - We write out the network for the complete sequence.
- For example, if the sequence we care about is a sentence of three words, the network would be unfolded into a 3-layer neural network.
  - One layer for each word.



Layers of Recurrent Neurons

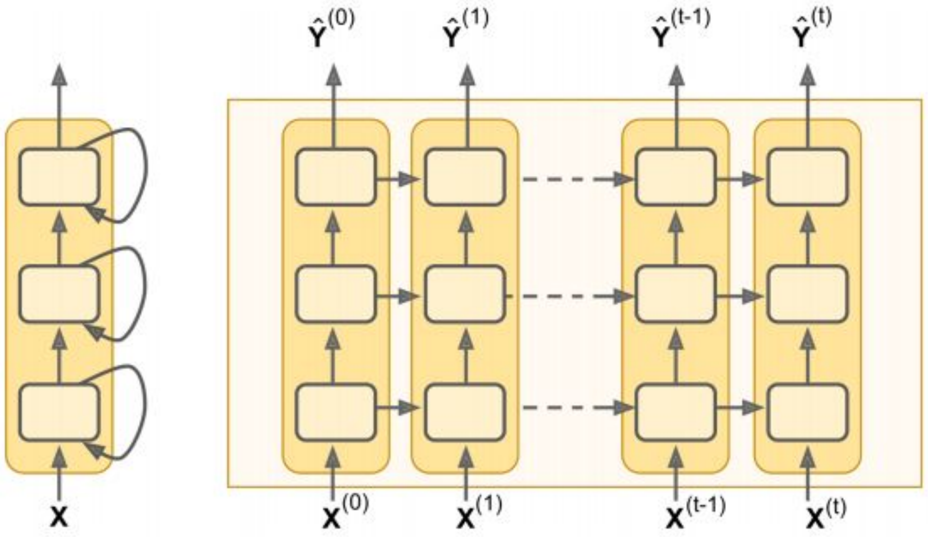
- At each time step t, every neuron of a layer receives both the input vector $x^{(t)}$ and the output vector from the previous time step $h^{(t-1)}$.





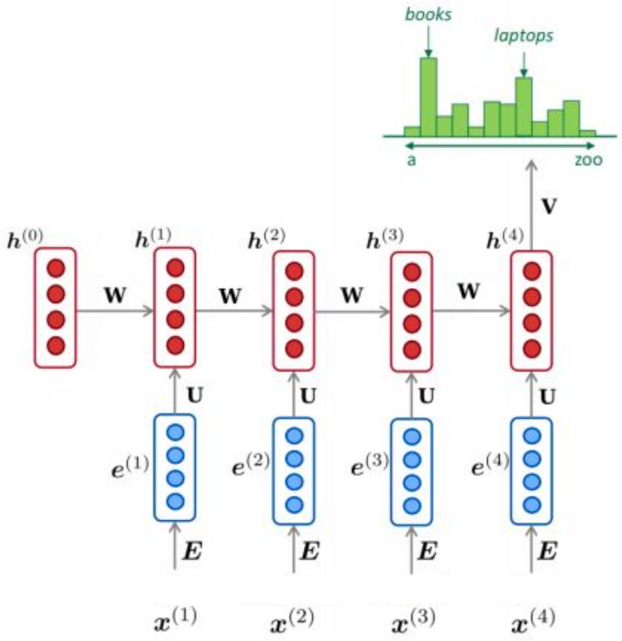
Deep RNN

- Stacking multiple layers of cells gives you a deep RNN.



Let’s Back to Language Model Example

- The equations for the RNN:
  - $h^{(t)} = \tanh(u^t e^{(t)} + wh^{(t-1)})$
  - $\hat{y}^{(t)} = \text{softmax}(vh^{(t)})$
- The output $\hat{y}^{(t)}$ is a vector of vocabulary size elements.
- Each element of $\hat{y}^{(t)}$ represents the probability of that word being the next word in the sentence.

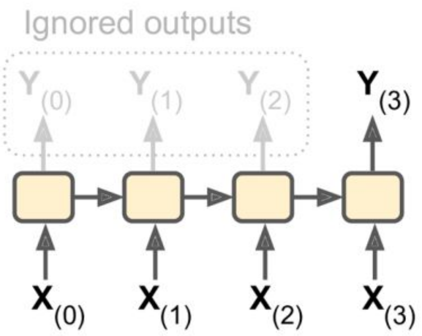


RNN Design Patterns

- Sequence-to-vector
- Vector-to-sequence
- Sequence-to-sequence
- Encoder-decoder

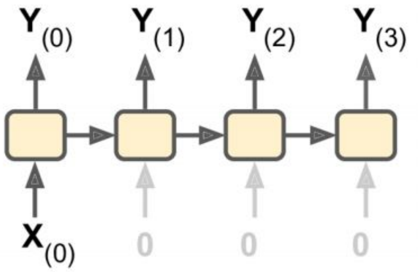
RNN Design Patterns

- Sequence-to-vector network: takes a sequence of inputs, and ignore all outputs except for the last one.
- E.g., you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score.



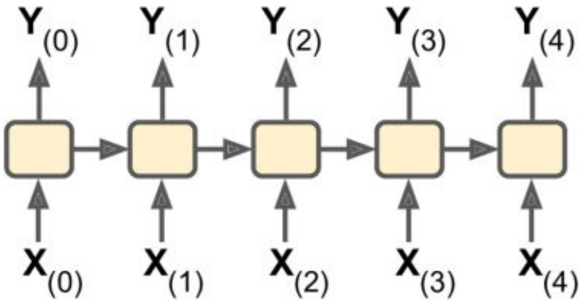
RNN Design Patterns

- Vector-to-sequence network: takes a single input at the first time step, and let it output a sequence.
- E.g., the input could be an image, and the output could be a caption for that image.



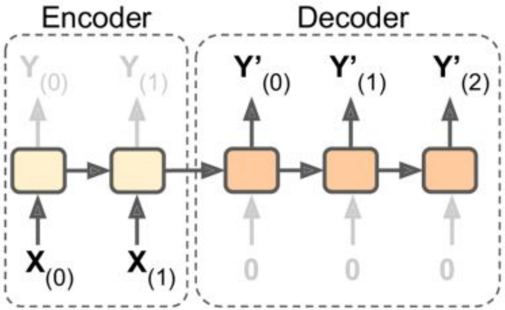
RNN Design Patterns

- Sequence-to-sequence network: takes a sequence of inputs and produce a sequence of outputs.
- Useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future.
- Here, both input sequences and output sequences have the same length.



RNN Design Patterns

- Encoder-decoder network: a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder).
- E.g., translating a sentence from one language to another.
- You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language.



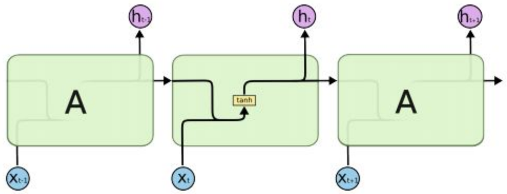
LSTMs

RNN Problems

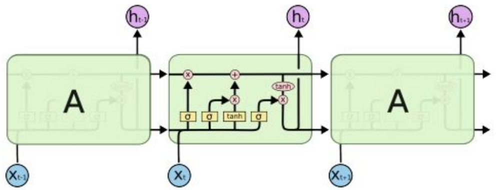
- Sometimes we only need to look at recent information to perform the present task.
  - E.g., predicting the next word based on the previous ones.
- In such cases, where the gap between the relevant information and the place that it’s needed is small, RNNs can learn to use the past information.
- But, as that gap grows, RNNs become unable to learn to connect the information.
- RNNs may suffer from the vanishing/exploding gradients problem.
- To solve these problem, long short-term memory (LSTM) have been introduced.
- In LSTM, the network can learn what to store and what to throw away.

Looking inside LSTM

- The LSTM cell looks exactly like a basic RNN cell.
- The repeating module in a standard RNN contains a single layer.



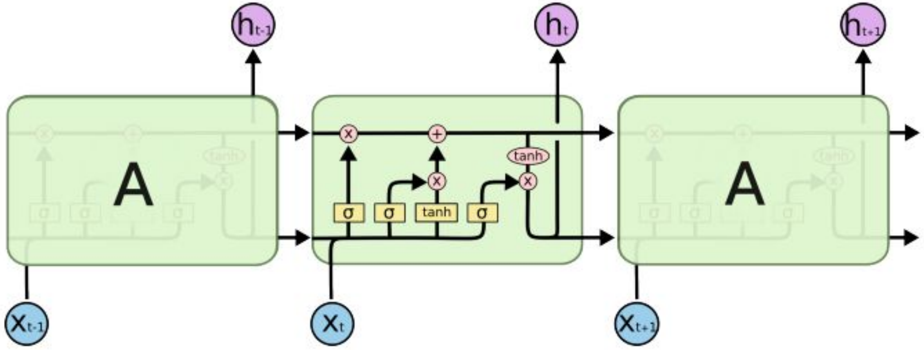
- The repeating module in an LSTM contains four interacting layers.



LSTM cell

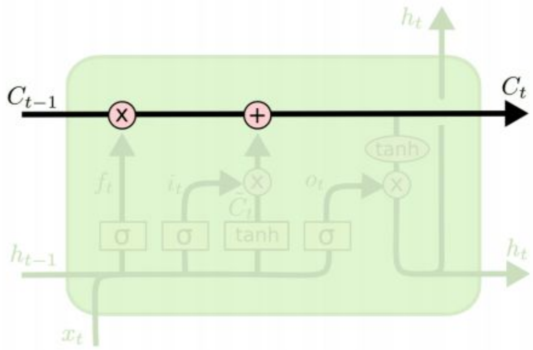
In LSTM state is split in two vectors:

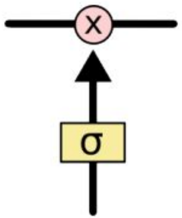
1. $h^{(t)}$ (h stands for hidden): the short-term state
2. $c^{(t)}$ (c stands for cell): the long-term state



LSTM cell

- The cell state (long-term state), the horizontal line on the top of the diagram.
- The LSTM can remove/add information to the cell state, regulated by three gates:
    - **Forget gate**, **input gate** and **output gate**





Autoencoders

An Example

- Which of them is easier to memorize?
- Seq1: $40, 27, 25, 36, 81, 57, 10, 73, 19, 68$
- Seq2: $50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$

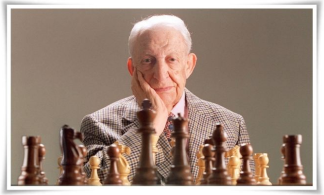
$$\text{Seq1} : 40, 27, 25, 36, 81, 57, 10, 73, 19, 68$$

$$\text{Seq2} : 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20$$

- Seq1 is shorter, so it should be easier.
- But, Seq2 follows two simple rules:
    - Even numbers are followed by their half.
    - Odd numbers are followed by their triple plus one.
- You don’t need pattern if you could quickly and easily memorize very long sequences
- But, it is hard to memorize long sequences that makes it useful to recognize patterns.

Memory

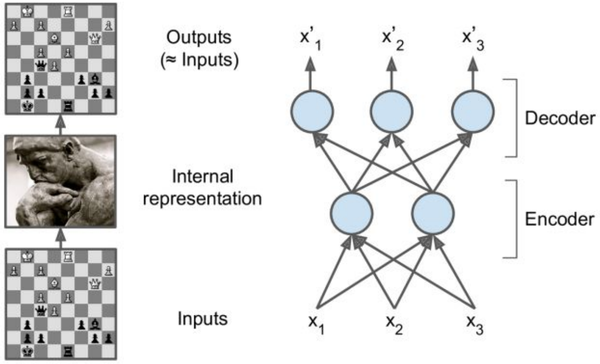
- 1970, W. Chase and H. Simon
- They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just 5 seconds.



- This was only the case when the pieces were placed in realistic positions, not when the pieces were placed randomly.
- Chess experts don’t have a much better memory than you and I.
- They just see chess patterns more easily due to their experience with the game. Patterns helps them store information efficiently.

Autoencoders

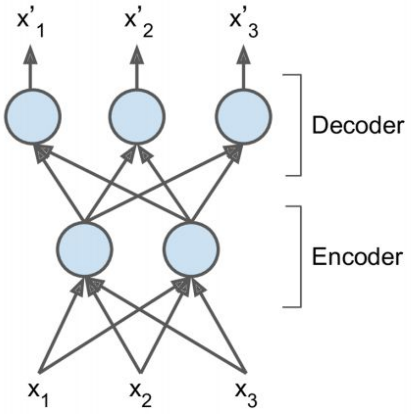
- Just like the chess players in this memory experiment.
- An autoencoder looks at the inputs, converts them to an efficient internal representation, and then spits out something that looks very close to the inputs.



Autoencoders

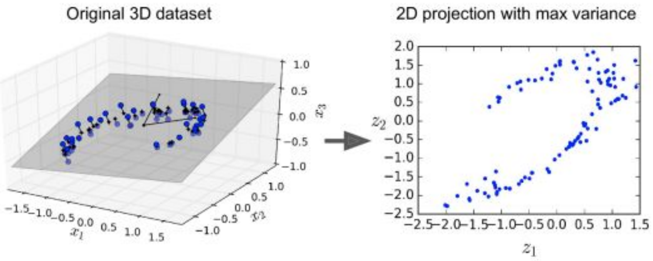
- An autoencoder is always composed of two parts.

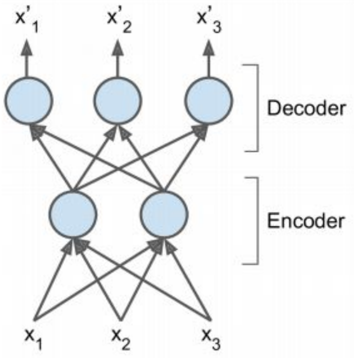
- An encoder (recognition network), $h = f(x)$
  Converts the inputs to an internal representation.
- A decoder (generative network), $r = g(h)$
  Converts the internal representation to the outputs.
- If an autoencoder learns to set $g(f(x)) = x$ everywhere, it is not especially useful, why?
  - Autoencoders are designed to be unable to learn to copy perfectly.
  - The models are forced to prioritize which aspects of the input should be copied, they often learn useful properties of the data.



Autoencoders

- Autoencoders are neural networks capable of learning efficient representations of the input data (called codings) without any supervision.
- Dimension reduction: these codings typically have a much lower dimensionality than the input data.



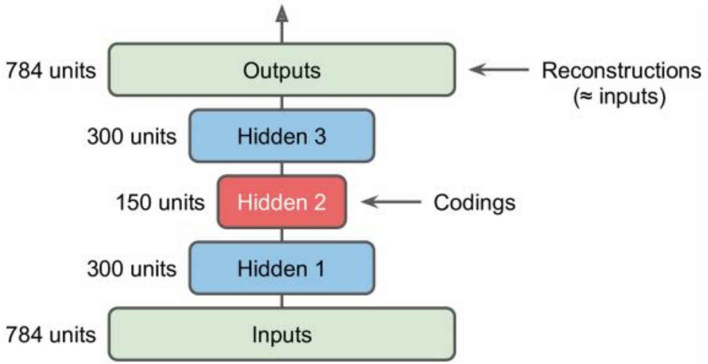


Two Types of Autoencoders

- Stacked autoencoders
- Variational autoencoders

Stacked autoencoders

- Stacked autoencoder: autoencoders with multiple hidden layers.
- Adding more layers helps the autoencoder learn more complex codings.
- The architecture is typically symmetrical with regards to the central hidden layer.



Variational Autoencoders

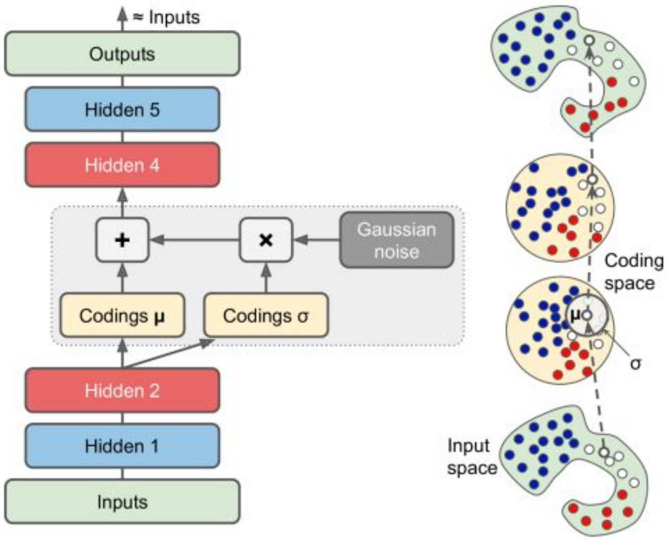
- Variational autoencoders are probabilistic autoencoders.
- Their outputs are partly determined by chance, even after training.
    - As opposed to denoising autoencoders, which use randomness only during training.
- They are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.

Variational Autoencoders

- Instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ.

- The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ.

- After that the decoder just decodes the sampled coding normally.



Variational Autoencoders

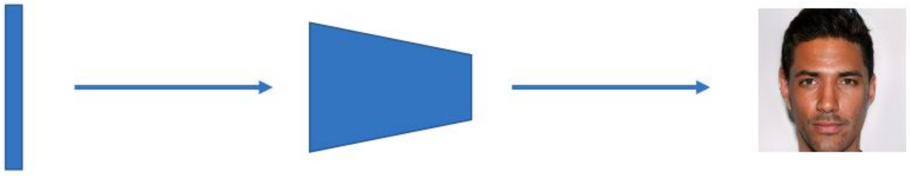
- The cost function is composed of two parts.

1. the usual reconstruction loss.
    - Pushes the autoencoder to reproduce its inputs.
    - Using cross-entropy
2. the latent loss
    - Pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution.
    - Using the KL divergence between the target distribution (the Gaussian distribution) and the actual distribution of the codings.
    - KL divergence measures the divergence between the two probabilities.

Generative adversarial networks

[CMPT 726: Machine Learning, Dr. Greg Mori]

GANs

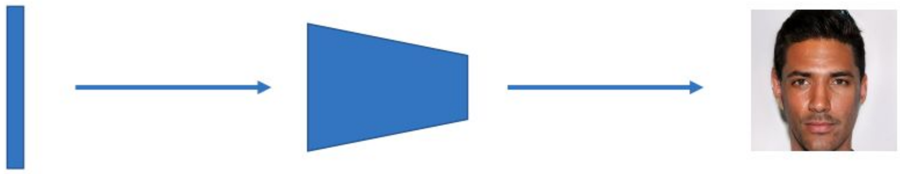


$\mathbf{z} \sim p(\mathbf{z})$ $\quad$ $G(\mathbf{z}, \theta_g)$ $\quad$ $\mathbf{x}$

Sample a random code
- Represents properties of the thing to generate

Generator maps a random code to an output
- Generator is a (deep, convolutional) neural network

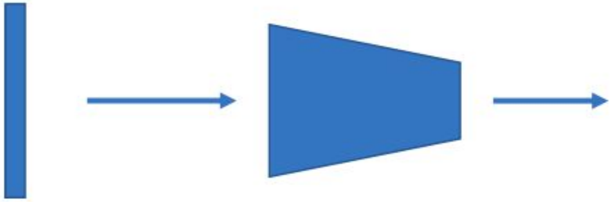
GANs



$\mathbf{z} \sim p(\mathbf{z})$ $\quad G(\mathbf{z}, \theta_g)$ $\quad \mathbf{x}$

How do we learn the parameters of the generator?
No labeled data?
But lots of real unlabeled data…

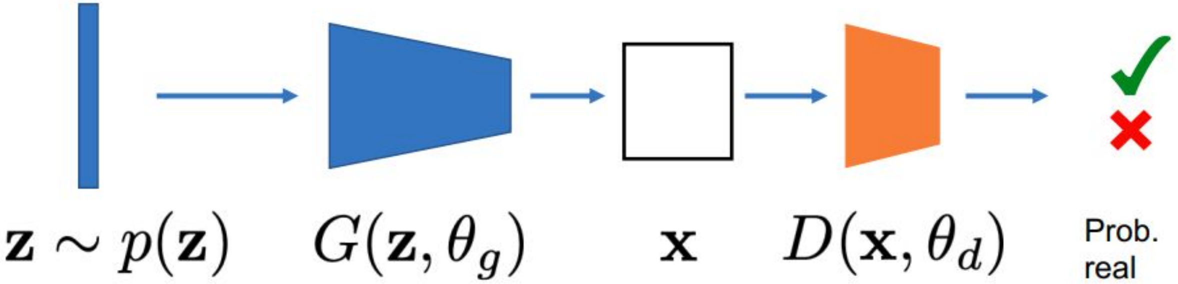
GANs



$\mathbf{z} \sim p(\mathbf{z})$ $\quad G(\mathbf{z}, \theta_g)$

But which face should we generate?
What is a good output?
What if we generate non-faces?
Overfitting?
Variety?

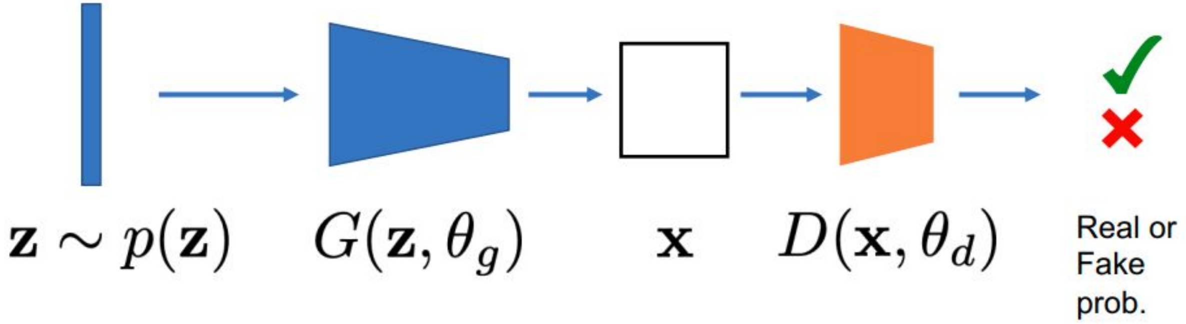
GANs



Given a discriminator, optimize the generator to fool the discriminator
- Generator should generate images that the discriminator thinks are real images, minimize wrt G:

$$\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z}, \theta_g), \theta_d))]$$

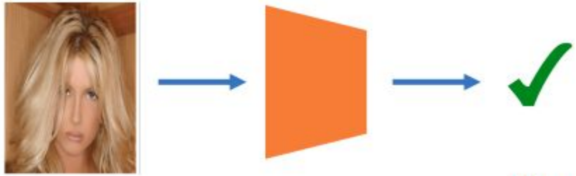
GANs



How do we train the discriminator?

GANs

$$\mathbf{x} \sim p_{data}(\mathbf{x})$$



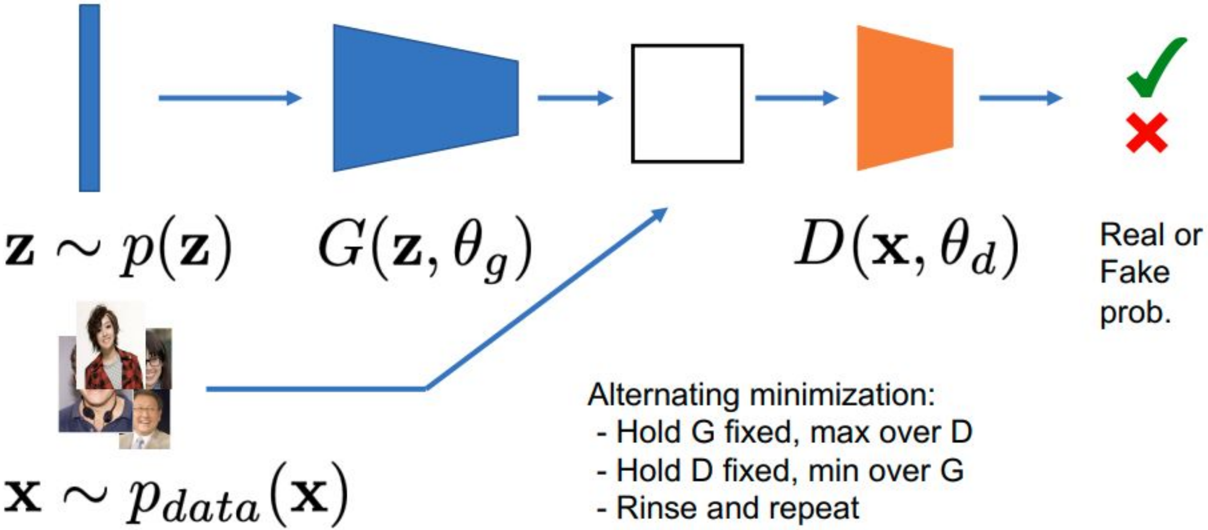
Should return real

$$D(\mathbf{x}, \theta_d)$$

Over the real images, make sure the discriminator thinks they are real
Maximize wrt. D:

$$\mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[\log D(\mathbf{x}, \theta_d)]$$

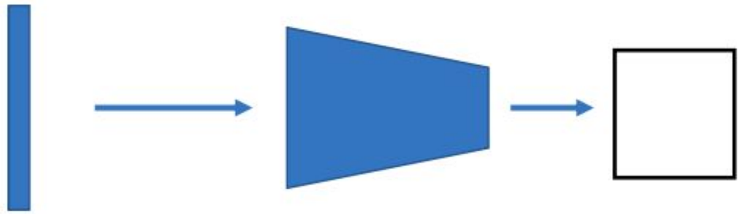
GANs



Alternating minimization:
- Hold G fixed, max over D
- Hold D fixed, min over G
- Rinse and repeat

$$\min_{\theta_g} \max_{\theta_d} \mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[\log D(\mathbf{x}, \theta_d)] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z}, \theta_g), \theta_d))]$$

GANs



$\mathbf{z} \sim p(\mathbf{z})$ $\quad$ $G(\mathbf{z}, \theta_g)$

Alternating minimization:
- Hold G fixed, max over D
- Hold D fixed, min over G
- Rinse and repeat

Once finished, discard the discriminator
Generator should generate realistic images for samples of z