THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Applied Machine Intelligence and Reinforcement Learning

Professor Hamza F. Alsarhan

SEAS 8505

Lecture 8

August 3, 2024

Welcome to SEAS Online at George Washington University

Class will begin shortly

Audio: To eliminate background noise, please be sure your audio is muted. To speak, please click the hand icon at the bottom of your screen (Raise Hand). When instructor calls on you, click microphone icon to unmute. When you've finished speaking, *be sure to mute yourself again*.

Chat: Please type your questions in Chat.

Recordings: As part of the educational support for students, we provide downloadable recordings of each class session to be used exclusively by registered students in that particular class for their own private use. Releasing these recordings is strictly prohibited.

Agenda

- RNN Limitations, Introduction to Attention and Transformers
- Representation Learning and Autoencoders
- Generative Learning with GANs
- Homework Overview

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Natural Language Processing with RNNs and Attention

Professor Hamza F. Alsarhan
SEAS 8505

Turing Test

- When Alan Turing imagined his famous Turing test in 1950, his objective was to evaluate a machine's ability to match human intelligence.
- He could have tested for many things, such as the ability to recognize cats in pictures, play chess, compose music, or escape a maze, but, interestingly, he chose a linguistic task.
- More specifically, he devised a chatbot capable of fooling its interlocutor into thinking it was human.
- This test highlights the fact that mastering language is arguably Homo sapiens' greatest cognitive ability.
- It turns out that research in Natural Language has proven invaluable for other applied machine learning domains.
- Other views:

✓ **Storytelling** (Winston): is a fundamental aspect of human communication and cognition, allowing us to organize and convey information, understand complex concepts, and make sense of the world around us.

✓ **Embodied Cognition:** posits that intelligence arises from interactions between an agent and its environment and other intelligent agents.

Return to RNNs

- Char-RNN: Introduction to character-level RNN, predicting subsequent characters in sentences.
- Text Generation: Utilizing Char-RNN for generating original text.
- Stateless RNN: Initial approach involves stateless RNN, learning from random text segments independently.
- Stateful RNN: Advancing to stateful RNN, preserving hidden states between iterations for capturing longer patterns.
- Sentiment Analysis: Transitioning to RNN-based sentiment analysis, treating sentences as sequences of words.
- Encoder-Decoder Architecture: Showcasing RNNs in encoder-decoder architecture for neural machine translation (NMT).
- Text Example: Demonstrating NMT with RNNs, translating English sentences into Spanish.

Char-RNN: Generating the next character in

Load the text corpus:

```
import tensorflow as tf

shakespeare_url = "https://homl.info/shakespeare"  # shortcut URL
filepath = tf.keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Train the RNN:

```
tf.random.set_seed(42)  # extra code - ensures reproducibility on CPU
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(input_dim=n_tokens, output_dim=16),
    tf.keras.layers.GRU(128, return_sequences=True),
    tf.keras.layers.Dense(n_tokens, activation="softmax")
])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model_ckpt = tf.keras.callbacks.ModelCheckpoint(
    "my_shakespeare_model", monitor="val_accuracy", save_best_only=True)
history = model.fit(train_set, validation_data=valid_set, epochs=10,
                    callbacks=[model_ckpt])
```

Guess the next Character:

```
y_proba = shakespeare_model.predict(["To be or not to b"])[0, -1]
y_pred = tf.argmax(y_proba)  # choose the most probable character ID
text_vec_layer.get_vocabulary()[y_pred + 2]
```

```
'e'
```

Generating Fake Shakespearean Text Blocks

- Greedy decoding: iterative prediction of the next character. Often results in repetitive sequences.
- Sampling the next character randomly based on the estimated probability distribution
  - tf.random.categorical() function facilitates this
  - It samples random class indices given the class log probabilities (logits)
- Temperature: To control the diversity of generated text, we can adjust a parameter called the temperature.
  - A low temperature favors high-probability characters, while a high temperature provides equal probabilities for all characters.
  - Lower temperatures are suitable for generating precise text, such as mathematical equations, while higher temperatures encourage creativity and diversity.

```
>>> tf.random.set_seed(42)
>>> print(extend_text("To be or not to be", temperature=0.01))
To be or not to be the duke
as it is a proper strange death,
and the
>>> print(extend_text("To be or not to be", temperature=1))
To be or not to behold?

second push:
gremio, lord all, a sistermen,
>>> print(extend_text("To be or not to be", temperature=100))
To be or not to bef ,mt'&o3fpadm!$
wh!nse?bws3est--vgerdjw?c-y-ewznq
```

Not very impressive!

Strategies for Improvement

- Nucleus Sampling: sample exclusively from the top k characters or from the smallest set of top characters whose combined probability surpasses a certain threshold.

- Beam Search: keeps track of a short list of the *k* most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the *k* most likely sentences. The parameter *k* is called the *beam width*. Discussed later

- Augmenting Model Architecture: by increasing the number of GRU layers and neurons per layer, extending the training duration, and incorporating regularization techniques as necessary

- Utilize a Stateful RNN: Discussed next

Stateful RNNs

- Stateless RNN: the model begins each training iteration with a hidden state of zeros, updates it at each time step, and discards it afterward.
- Previous model struggled with learning patterns longer than a given length, typically around 100 characters.
  - While enlarging this window may seem beneficial, it could also intensify training challenges, as even LSTM and GRU cells face limitations with very lengthy sequences
- Stateful RNN: retains final state after processing a training batch and uses it as the initial state for the next batch
  - Allows the model to learn long-term patterns despite backpropagating through short sequences
  - This requires special handling of batches: sequential and nonoverlapping input sequences for each batch.
  - Set the “stateful” argument to TRUE for each layer and specify the batch_input_shape argument in the first layer to indicate the batch size.
  - Reset the states to the initial state before proceeding to the beginning of the text
  - After training, the stateful model can only be used for predictions with batches of the same size as those used during training. To alleviate this restriction, an identical stateless model can be created, and the stateful model's weights can be copied to this model.

Sentiment Analysis

- If MNIST is the “hello world” of computer vision, then the **IMDb reviews dataset** is the “hello world” of natural language processing: it consists of 50,000 movie reviews in English (25,000 for training, 25,000 for testing) extracted from the famous Internet Movie Database, along with a simple binary target for each review indicating whether it is negative (0) or positive (1).
- Just like MNIST, the IMDb reviews dataset is popular for good reasons: it is simple enough to be tackled on a laptop in a reasonable amount of time, but challenging enough to be fun and rewarding.
- All punctuation was removed, and then words were converted to lowercase, split by spaces, and indexed by frequency (so low integers correspond to frequent words).
- The integers 0, 1, and 2 are special: they represent the padding token, the start-of-sequence (SSS) token, and unknown words, respectively.

Application of RNNs for Sentiment Analysis

Text Preprocessing

- Split the data into training, validation, and testing sets.
- Ensure proper batching and shuffling for efficient training.
- TextVectorization layer employed for word-level tokenization.
- Considerations for tokenization methods, including subword tokenization techniques like byte pair encoding (BPE) and WordPiece.
- Adapt TextVectorization layer to the training set with limited vocabulary size.

Model Building and Training

- Constructing the sentiment analysis model with TextVectorization, Embedding, GRU, and Dense layers.
- Model compilation and specification of loss function and optimizer.
- Training the model on the prepared dataset for a few epochs.

Challenges and Solutions

- Issues arise due to varying lengths of reviews leading to padding tokens.
- Solutions involve ensuring equal-length sentences in batches or incorporating masking to ignore padding tokens.

Reusing Pretrained Embeddings and Language Models

- **Leveraging Pretrained Word Embeddings**
  - Limited data, yet effective embeddings
  - Possible enhancement with larger datasets
  - Utilize embeddings trained on different corpora
- **Limitations of Pretrained Word Embeddings**
  - Fixed representation regardless of context
  - Introduction of Contextualized Word Embeddings
- **Contextualized Word Embeddings: ELMo**
  - Introduced by Matthew Peters in 2018
  - Learns from internal states of bidirectional language models
  - Addresses limitations of fixed embeddings
- **Unsupervised Pretraining in NLP: ULMFiT**
  - Introduced by Jeremy Howard and Sebastian Ruder
  - LSTM language model trained on large corpus
  - Outperforms state of the art on text classification tasks

- **Evolution: Reusing Pretrained Language Models**
  - ULMFiT marks the shift in NLP
  - Pretrained language models become standard
- **Application: Building a Classifier with Universal Sentence Encoder**
  - Utilizes Universal Sentence Encoder from Google
  - Based on transformer architecture
  - Achieves validation accuracy of over 90%
- **Fine-tuning Pretrained Models**
  - Set **trainable=True** for fine-tuning
  - Ensures the pretrained Universal Sentence Encoder is fine-tuned during training

```
import os
import tensorflow_hub as hub

os.environ["TFHUB_CACHE_DIR"] = "my_tfhub_cache"
tf.random.set_seed(42)  # extra code – ensures reproducibility on CPU
model = tf.keras.Sequential([
    hub.KerasLayer("https://tfhub.dev/google/universal-sentence-encoder/4",
                   trainable=True, dtype=tf.string, input_shape=[]),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.fit(train_set, validation_data=valid_set, epochs=10)
```

An Encoder–Decoder Network for Neural Machine Translation

- Figure 16-3 is a simple neural machine translation model that will translate English sentences to Spanish.
- English sentences are fed to the encoder, and the decoder outputs the Spanish translations.
- Note that the Spanish translations are also used as inputs to the decoder, but shifted back by one step.
  - The decoder is given as input the word that it should have output at the previous step (regardless of what it actually output).
  - For the very first word, the decoder is given the **start-of-sequence (SOS) token**, and the decoder is expected to end the sentence with an **end-of-sequence (EOS) token**.

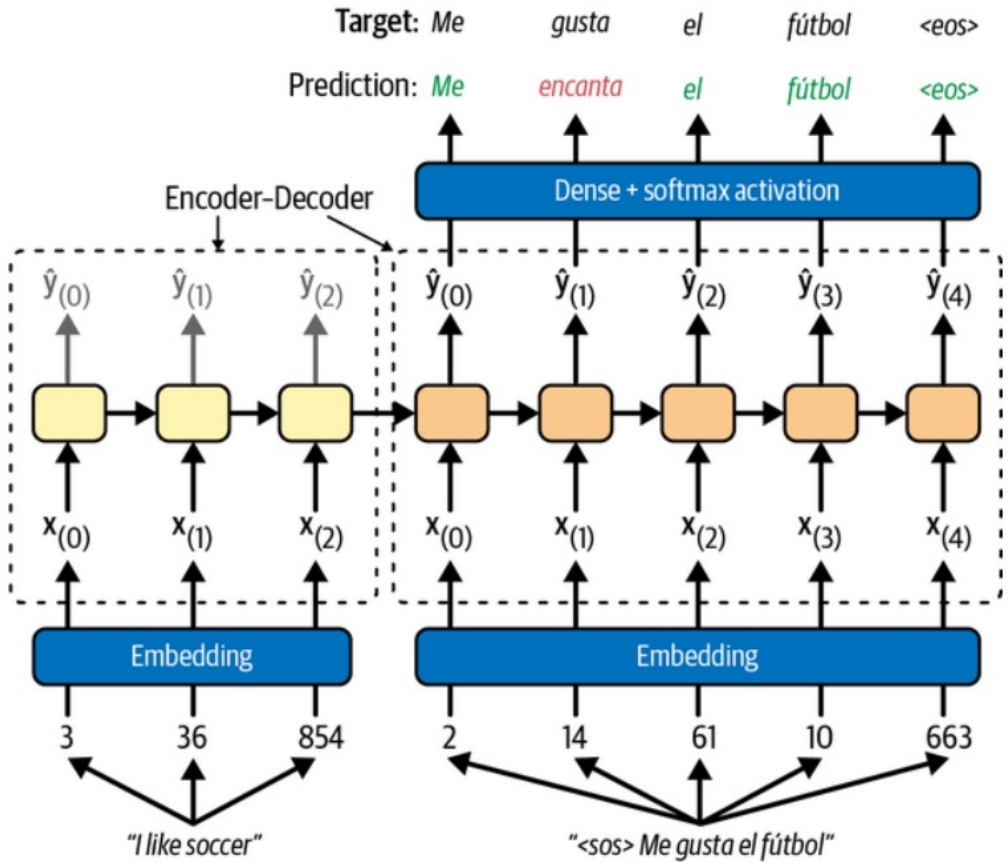


Figure 16-3. A simple machine translation model

An Encoder–Decoder Network for Neural Machine Translation

- Note that at inference time (after training), you will not have the target sentence to feed to the decoder.
- Instead, you need to feed it the word that it has just output at the previous step, as shown in the Figure.

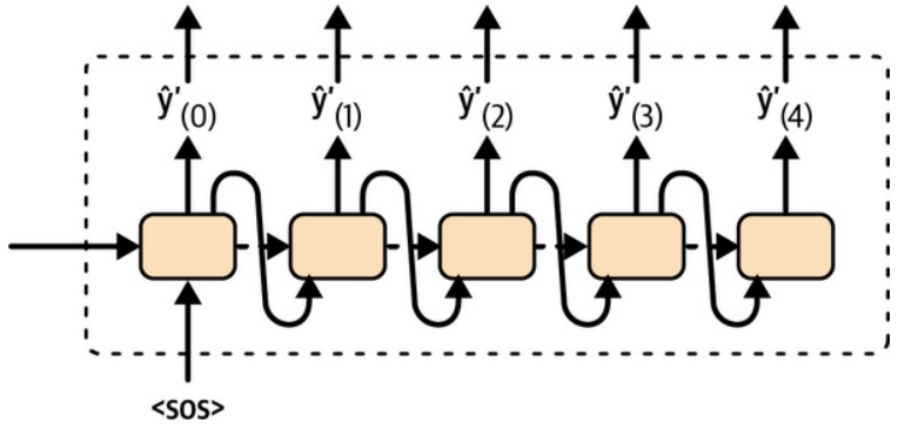


Figure 16-4. At inference time, the decoder is fed as input the word it just output at the previous time step

```
url = "https://storage.googleapis.com/download.tensorflow.org/data/spa-eng.zip"
path = tf.keras.utils.get_file("spa-eng.zip", origin=url, cache_dir="datasets",
                               extract=True)
text = (Path(path).with_name("spa-eng") / "spa.txt").read_text()
```

```
import numpy as np

text = text.replace("¡", "").replace("¿", "")
pairs = [line.split("\t") for line in text.splitlines()]
np.random.seed(42)  # extra code – ensures reproducibility on CPU
np.random.shuffle(pairs)
sentences_en, sentences_es = zip(*pairs)  # separates the pairs into 2 lists
```

```
for i in range(3):
    print(sentences_en[i], "=>", sentences_es[i])
```

How boring! => Qué aburrimiento!
I love sports. => Adoro el deporte.
Would you like to swap jobs? => Te gustaría que intercambiemos los trabajos?

An Encoder–Decoder Network for Neural Machine Translation

Let's create two TextVectorization layers – one per layer:

```
vocab_size = 1000
max_length = 50
text_vec_layer_en = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_es = tf.keras.layers.TextVectorization(
    vocab_size, output_sequence_length=max_length)
text_vec_layer_en.adapt(sentences_en)
text_vec_layer_es.adapt([f"startofseq {s} endofseq" for s in sentences_es])
```

Let's inspect the first 10 tokens in both vocabularies. They start with the padding token, the unknown token, the SOS and EOS tokens (only in the Spanish vocabulary), then the actual words, sorted by decreasing frequency:

```
text_vec_layer_en.get_vocabulary()[:10]
```

```
['', '[UNK]', 'the', 'i', 'to', 'you', 'tom', 'a', 'is', 'he']
```

```
text_vec_layer_es.get_vocabulary()[:10]
```

```
['', '[UNK]', 'startofseq', 'endofseq', 'de', 'que', 'a', 'no', 'tom', 'la']
```

Hands-On Machine Learning

Create the training set and the validation set. Use the first 100,000 sentence pairs for training, and the rest for validation. The decoder's inputs are the Spanish sentences plus an SOS token prefix. The targets are the Spanish sentences plus an EOS suffix:

```
X_train = tf.constant(sentences_en[:100_000])
X_valid = tf.constant(sentences_en[100_000:])
X_train_dec = tf.constant([f"startofseq {s}" for s in sentences_es[:100_000]])
X_valid_dec = tf.constant([f"startofseq {s}" for s in sentences_es[100_000:]])
Y_train = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[:100_000]])
Y_valid = text_vec_layer_es([f"{s} endofseq" for s in sentences_es[100_000:]])
```

Build the translation model using the functional API. It requires two text inputs—one for the encoder and one for the decoder:

```
encoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
decoder_inputs = tf.keras.layers.Input(shape=[], dtype=tf.string)
```

Encode sentences using the TextVectorization layers prepared earlier, followed by an Embedding layer for each language, with mask_zero=True to ensure masking is handled automatically.

```
embed_size = 128
encoder_input_ids = text_vec_layer_en(encoder_inputs)
decoder_input_ids = text_vec_layer_es(decoder_inputs)
encoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
decoder_embedding_layer = tf.keras.layers.Embedding(vocab_size, embed_size,
                                                    mask_zero=True)
encoder_embeddings = encoder_embedding_layer(encoder_input_ids)
decoder_embeddings = decoder_embedding_layer(decoder_input_ids)
```

An Encoder–Decoder Network for Neural Machine Translation

- There are a few more steps provided in the textbook
- You can't just call model.predict() because the decoder expects as input the word that was predicted at the previous time step
- We need a "Translate Utility" that calls the model multiple times, predicting one extra word at each round
- The results are summarized on the right
- NOTE: Longer sentences can pose a problem
- To help with this we are going to look at using Bidirectional Recurrent Layers

```
def translate(sentence_en):
    translation = ""
    for word_idx in range(max_length):
        X = np.array([sentence_en])  # encoder input
        X_dec = np.array(["startofseq " + translation])  # decoder input
        y_proba = model.predict((X, X_dec))[0, word_idx]  # last token's probas
        predicted_word_id = np.argmax(y_proba)
        predicted_word = text_vec_layer_es.get_vocabulary()[predicted_word_id]
        if predicted_word == "endofseq":
            break
        translation += " " + predicted_word
    return translation.strip()
```

```
translate("I like soccer")
```

```
'me gusta el fútbol'
```

Nice! However, the model struggles with longer sentences:

```
translate("I like soccer and also going to the beach")
```

```
'me gusta el fútbol y a veces mismo al bus'
```

Bidirectional RNNs

- At each time step, a regular recurrent layer only looks at past and present inputs before generating its output.
  - It is “causal,” meaning it cannot look into the future
- For example, consider the phrases “the Queen of the United Kingdom,” “the queen of hearts,” and “the queen bee”: to properly encode the word “queen,” you need to look ahead.
  - To implement this, run two recurrent layers on the same inputs, one reading the words from left to right and the other reading them from right to left.
  - Then simply combine their outputs at each time step, typically by concatenating them.
  - This is called a bidirectional recurrent layer

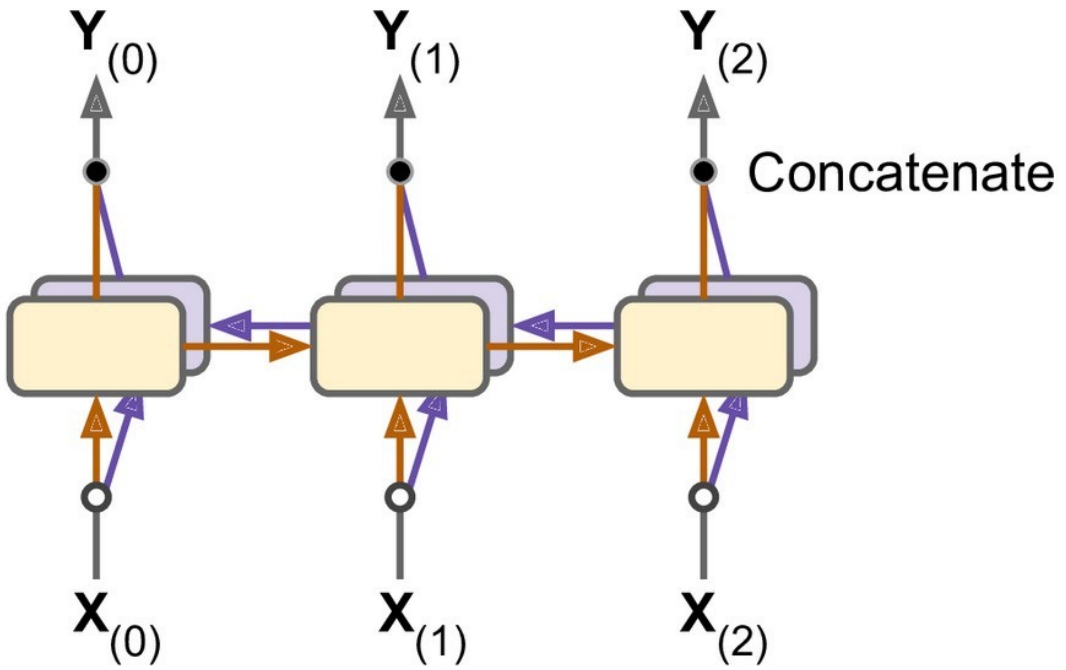
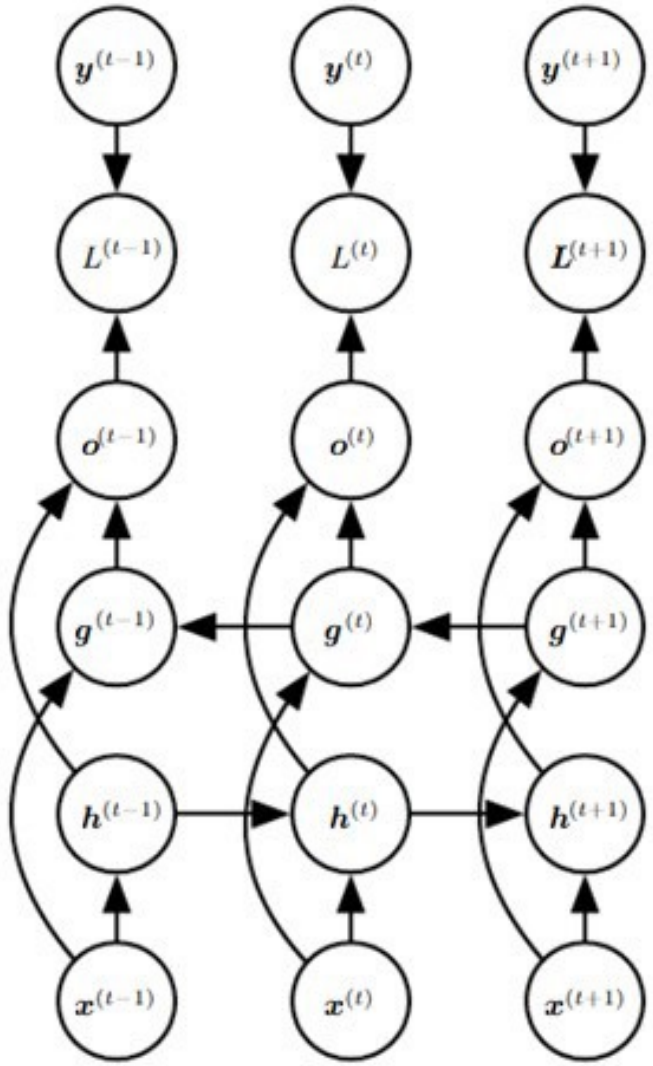


Figure 16-5. A bidirectional recurrent layer

Bidirectional RNNs

- For sequences other than time series (e.g. text), it is often the case that an RNN model can perform better if it not only processes sequence from start to end, but also backwards.
- For example, to predict the next word in a sentence, it is often useful to have the **context around the word**, not only just the words that come before it.

- ***h*** recurrence propagates information forward in time
- ***g*** recurrence propagates information backward in time



Based on Deep Learning (Goodfellow, Bengio, Courville), Chapter 10, Figure 10.11
https://www.tensorflow.org/guide/keras/rnn

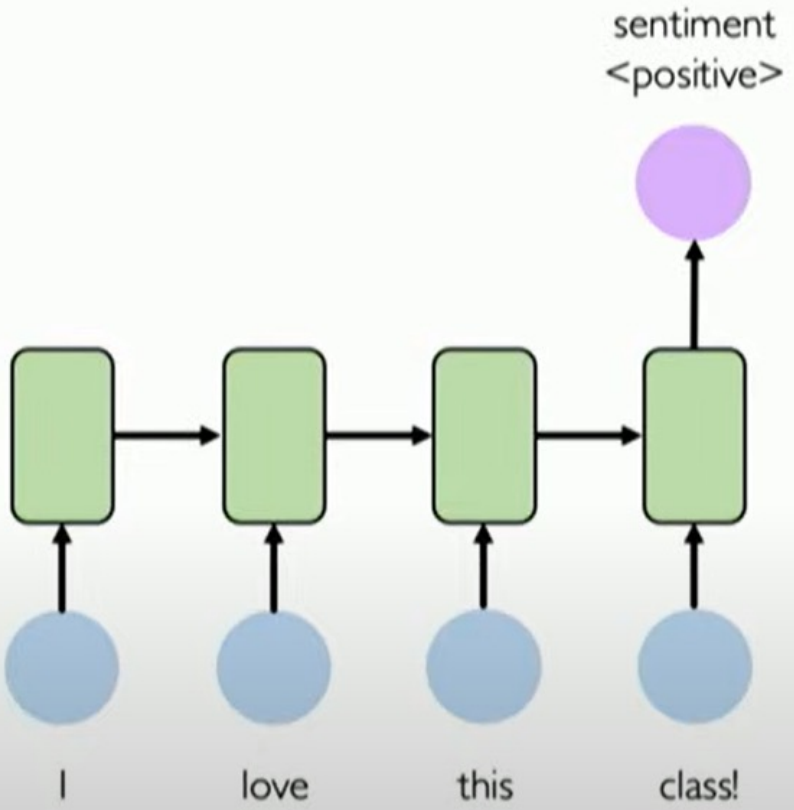
Beam Search

- Suppose you have trained an encoder–decoder model, and you use it to translate the sentence “I like soccer” to Spanish.
- You are hoping that it will output the proper translation “me gusta el fútbol”, but unfortunately it outputs “me gustan los jugadores”, which means “I like the players”.
- Looking at the training set, you notice many sentences such as “I like cars”, which translates to “me gustan los autos”, so it wasn’t absurd for the model to output “me gustan los” after seeing “I like”. Unfortunately, in this case it was a mistake since “soccer” is singular.
- The model could not go back and fix it, so it tried to complete the sentence as best it could, in this case using the word “jugadores”.
- How can we give the model a chance to go back and fix mistakes it made earlier?
- One of the most common solutions is ***beam search***: it keeps track of a short list of the k most promising sentences (say, the top three), and at each decoder step it tries to extend them by one word, keeping only the k most likely sentences.
- The parameter k is called the *beam width*.

Figure 16-6. Beam search, with a beam width of 3

- With all this, you can get reasonably good translations for fairly short sentences.
- Unfortunately, this model will be really bad at translating long sentences.
- Once again, the problem comes from the limited short-term memory of RNNs.
- *Attention mechanisms* are the game-changing innovation that addressed this problem.

Limitations of Recurrent Models



Limitations of RNNs

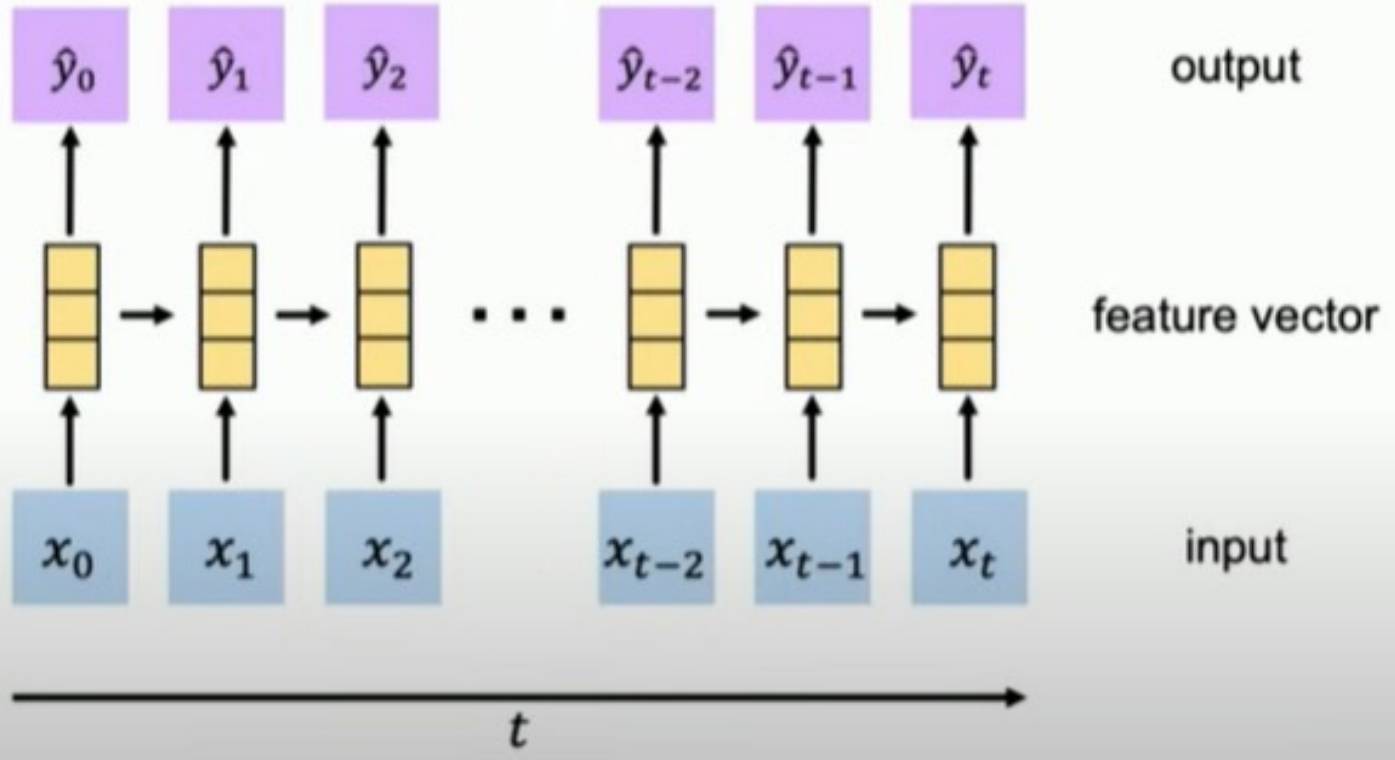
- Encoding bottleneck
- Slow, no parallelization
- Not long memory

Goal of Sequence Modeling

Limitations of RNNs

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

RNNs: recurrence to model sequence dependencies

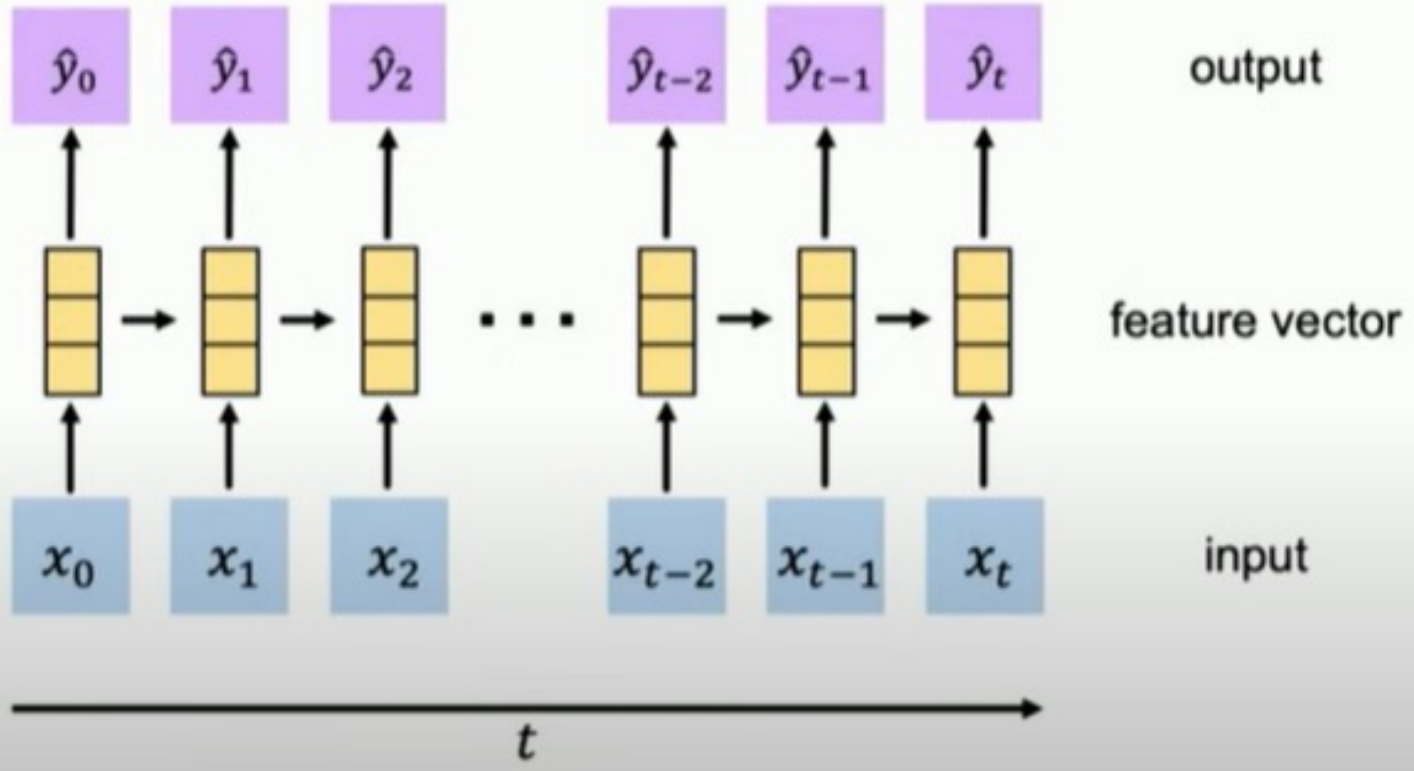


Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

Desired Capabilities

- Continuous stream
- Parallelization
- Long memory

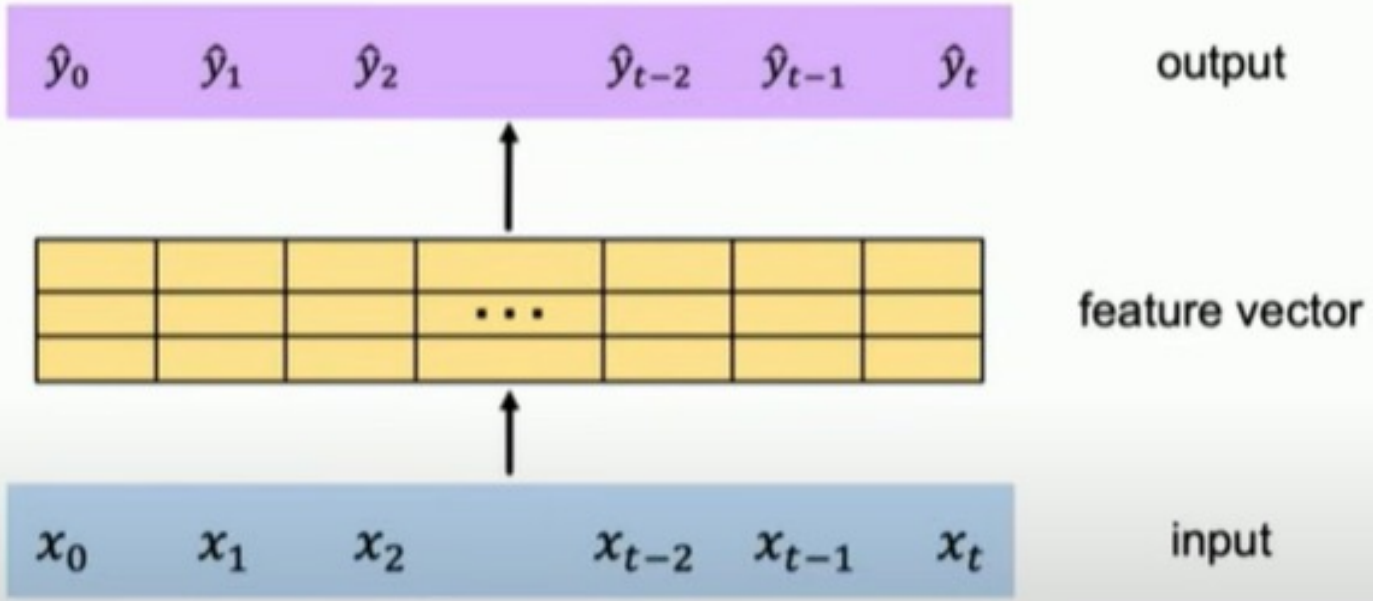


Goal of Sequence Modeling

Idea 1: Feed everything into dense network

- ✓ No recurrence
- ✗ Not scalable
- ✗ No order
- ✗ No long memory

Can we eliminate the need for recurrence entirely?



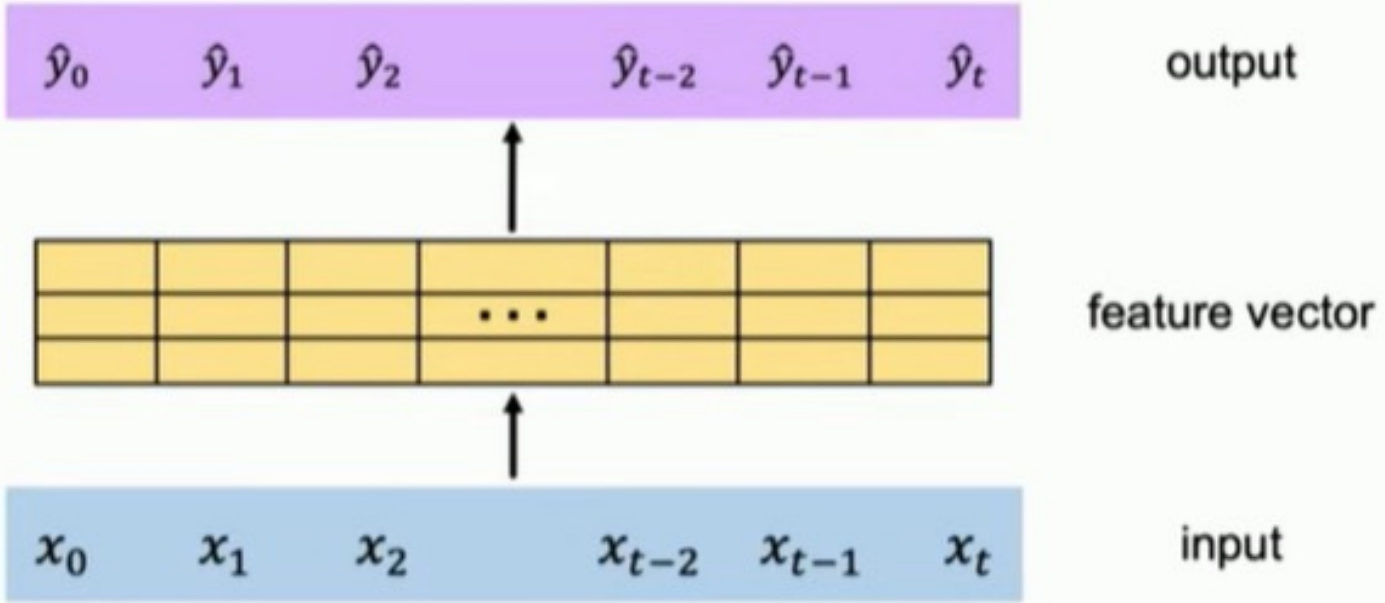
Goal of Sequence Modeling

Idea 1: Feed everything into dense network

- ✓ No recurrence
- ✗ Not scalable
- ✗ No order
- ✗ No long memory

Idea: Identify and attend to what's important

Can we eliminate the need for recurrence entirely?



Attention Mechanisms

- Consider the path from the word “soccer” to its translation “fútbol” in Figure 16-3: it is quite long!
  - A representation of this word (along with all the other words) needs to be carried over many steps before it is used
- At the time step where the decoder needs to output the word “fútbol” it will focus its attention on the word “soccer”
- Attention is a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step
  - The path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact
  - Attention mechanisms revolutionized neural machine translation (and NLP in general), allowing a significant improvement in the state of the art, especially for long sentences (over 30 words)

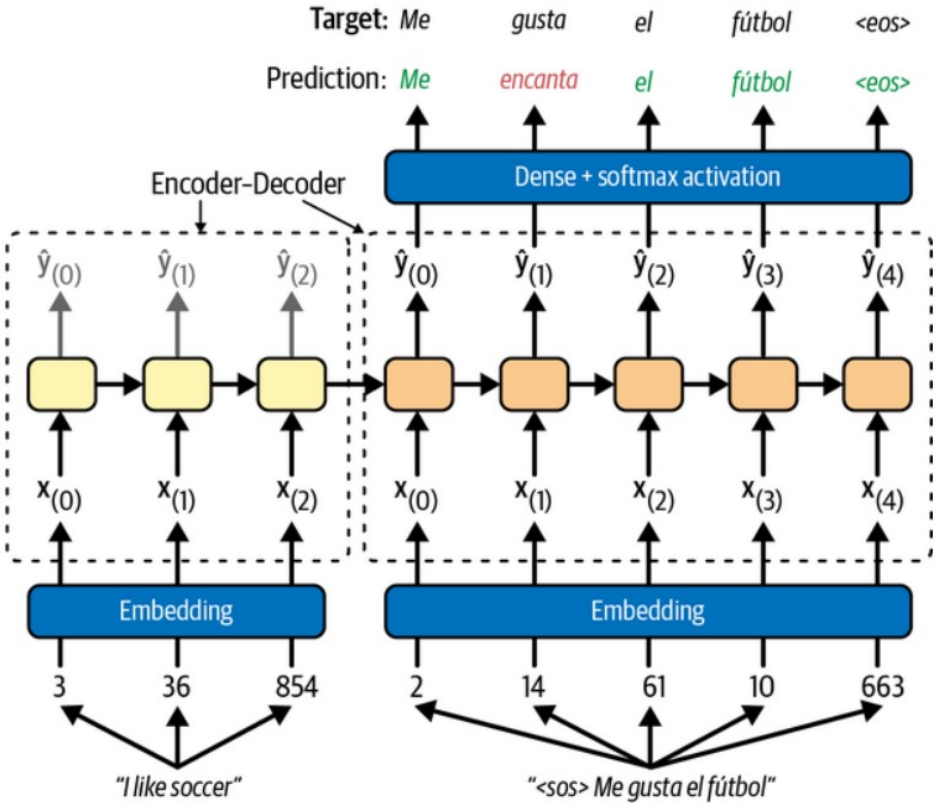


Figure 16-3. A simple machine translation model

Encoder-Decoder with Attention Mechanism

- Encoder–decoder model with an added attention mechanism.
- On the left, you have the encoder and the decoder.
- Instead of just sending the encoder's final hidden state to the decoder, as well as the previous target word at each step, we now send all of the encoder's outputs to the decoder as well.
- Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs.
- This determines which words it will focus on at this step.
- The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step.
- The rest of the decoder operates as usual: it receives the target word from the previous time step (or at inference time, the output from the previous time step).

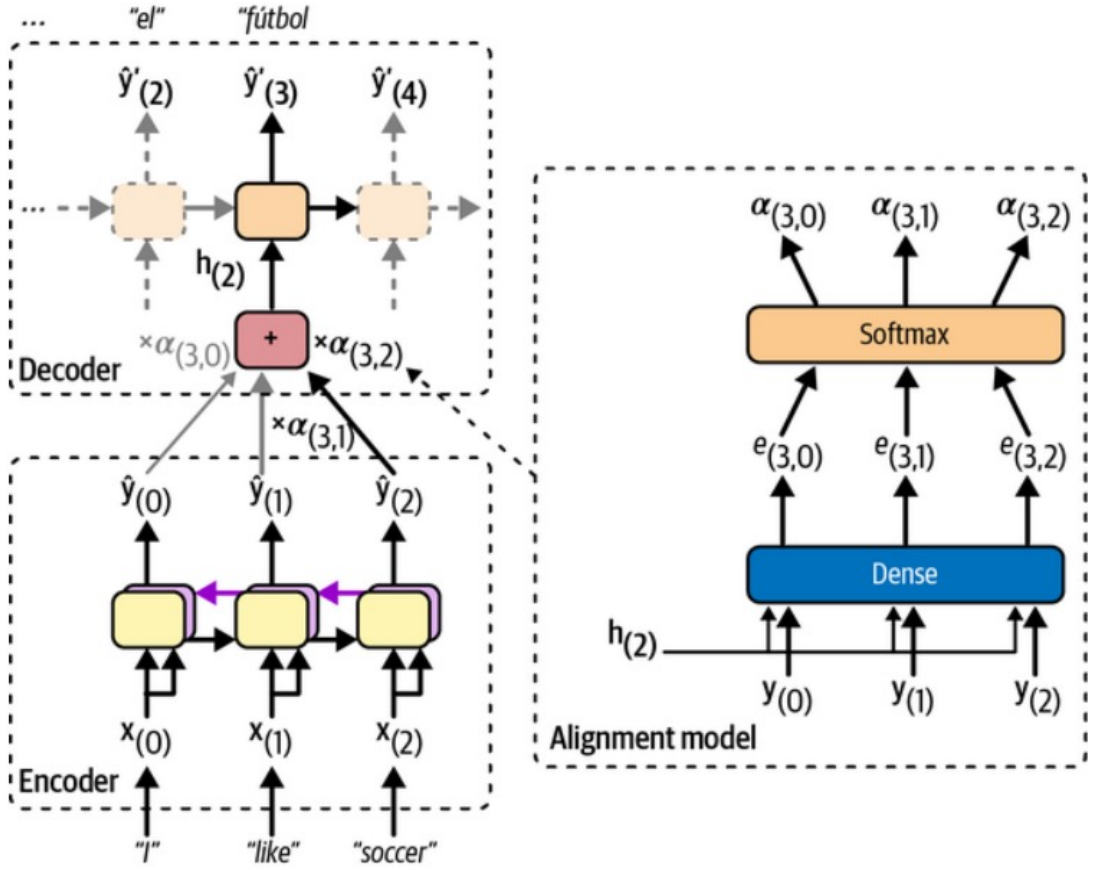


Figure 16-7. Neural machine translation using an encoder–decoder network with an attention model

Attention Is All You Need: The Transformer Architecture

- In a groundbreaking 2017 paper, a team of Google researchers suggested that “Attention Is All You Need.”
- They managed to create an architecture called the Transformer, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers, just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces).
- As an extra bonus, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

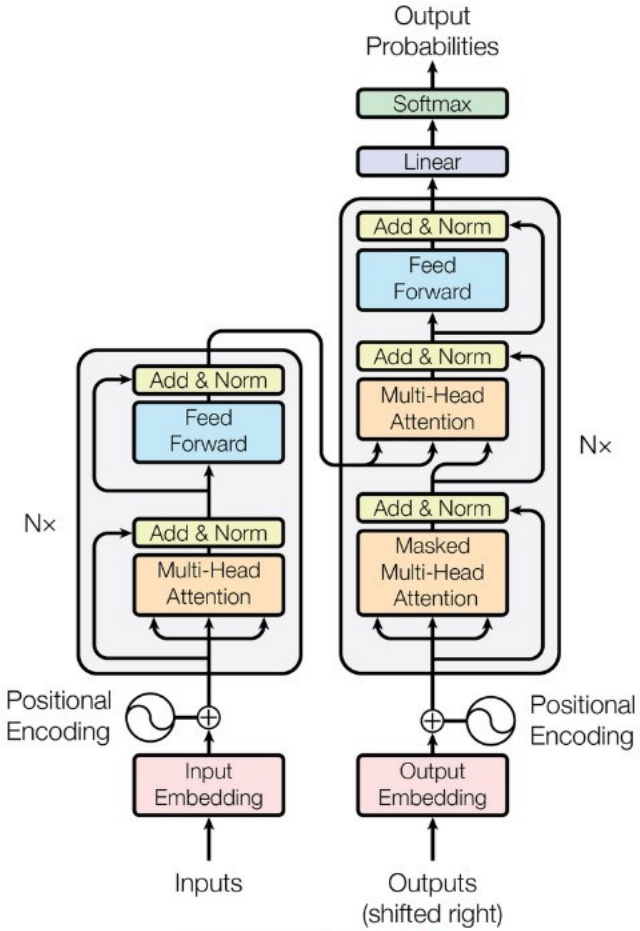


Figure 16-8. The Transformer architecture[22]

Intuition Behind Self-Attention

Attending to the most important parts of an input.

1. Identify which parts to attend to
2. Extract the features with high attention

Similar to a search problem!

Transformer Operation

- If you use the transformer for NMT, then during training you must feed the English sentences to the encoder and the corresponding Spanish translations to the decoder, with an extra SOS token inserted at the start of each sentence.
- At inference time, you must call the transformer multiple times, producing the translations one word at a time and feeding the partial translations to the decoder at each round, just like we did earlier in the translate() function.
- The **encoder's role** is to gradually transform the inputs—word representations of the English sentence—until each word's representation perfectly captures the meaning of the word, in the context of the sentence.
  - For example, if you feed the encoder with the sentence "I like soccer", then the word "like" will start off with a rather vague representation, since this word could mean different things in different contexts: think of "I like soccer" versus "It's like that".
- But after going through the encoder, the word's representation should capture the correct meaning of "like" in the given sentence (i.e., to be fond of), as well as any other information that may be required for translation (e.g., it's a verb).

Transformer Operation

- The **decoder's role** is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation.
  - For example, if the sentence to translate is "I like soccer", and the decoder's input sentence is "<SOS> me gusta el fútbol", then after going through the decoder, the word representation of the word "el" will end up transformed into a representation of the word "fútbol". Similarly, the representation of the word "fútbol" will be transformed into a representation of the EOS token.
  - After going through the decoder, each word representation goes through a final Dense layer with a softmax activation function, which will hopefully output a high probability for the correct next word and a low probability for all other words. The predicted sentence should be "me gusta el fútbol <EOS>".

A Closer Look at the Transformer

- First, notice that both the encoder and the decoder contain modules that are stacked *N* times. In the paper, *N* = 6.
- The final outputs of the whole encoder stack are fed to the decoder at each of these *N* levels.
- Zooming in, you can see that you are already familiar with most components:
  - There are two embedding layers; several skip connections, each of them followed by a layer normalization layer; several feedforward modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function); and finally the output layer is a dense layer using the softmax activation function.
  - You can also sprinkle a bit of dropout after the attention layers and the feedforward modules, if needed.

  NOTE: The first two arrows going into each multi-head attention layer in Figure 16-8 represent the keys and values, and the third arrow represents the queries. In the self-attention layers, all three are equal to the word representations output by the previous layer, while in the decoder's upper attention layer, the keys and values are equal to the encoder's final word representations, and the queries are equal to the word representations output by the previous layer.

But How Can We Translate a Sentence by Looking at Words Completely Separately?

Answer: We can't, so that's where the new components come in:

- The encoder's *multi-head attention* layer updates each word representation by attending to (i.e., paying attention to) all other words in the same sentence. That's where the vague representation of the word "like" becomes a richer and more accurate representation, capturing its precise meaning in the given sentence. We will discuss exactly how this works shortly.

- The decoder's *masked multi-head attention* layer does the same thing, but when it processes a word, it doesn't attend to words located after it: it's a causal layer. For example, when it processes the word "gusta", it only attends to the words "<SOS> me gusta", and it ignores the words "el fútbol" (or else that would be cheating).

- The decoder's upper *multi-head attention* layer is where the decoder pays attention to the words in the English sentence. This is called ***cross*-attention,** not ***self*-attention** in this case. For example, the decoder will probably pay close attention to the word "soccer" when it processes the word "el" and transforms its representation into a representation of the word "fútbol".

Positional Encodings

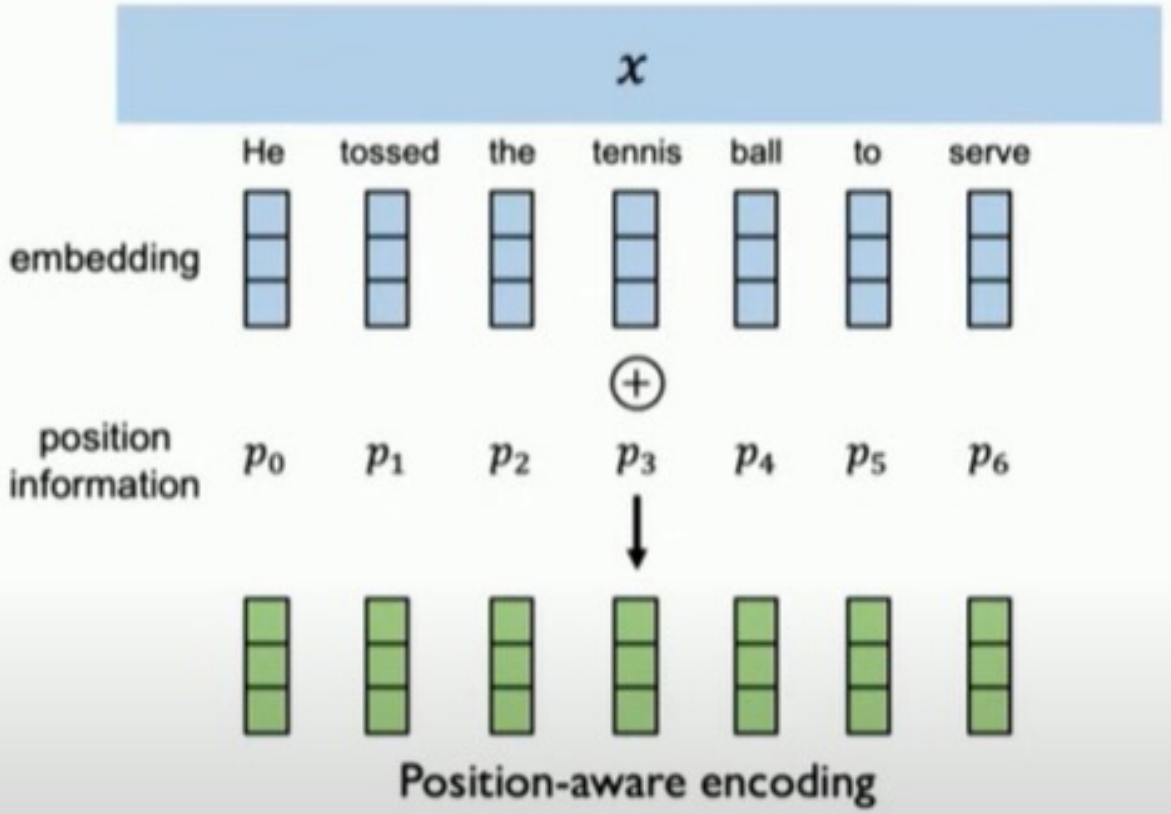
Positional Encodings are dense vectors (much like word embeddings) that represent the position of each word in the sentence.

- The n^{th} positional encoding is added to the word embedding of the n^{th} word in each sentence. This is needed because all layers in the transformer architecture ignore word positions:
- Without positional encodings, you could shuffle the input sequences, and it would just shuffle the output sequences in the same way.
- Obviously, the order of words matters, which is why we need to give positional information to the transformer somehow: adding positional encodings to the word representations is a good way to achieve this.
- A positional encoding is a dense vector that encodes the position of a word within a sentence*: the ith positional encoding is simply added to the word embedding of the ith word in the sentence.*

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract query, key, value for search
3. Compute attention weighting
4. Extract features with high attention



Data is fed in all at once! Need to encode position information to understand order.

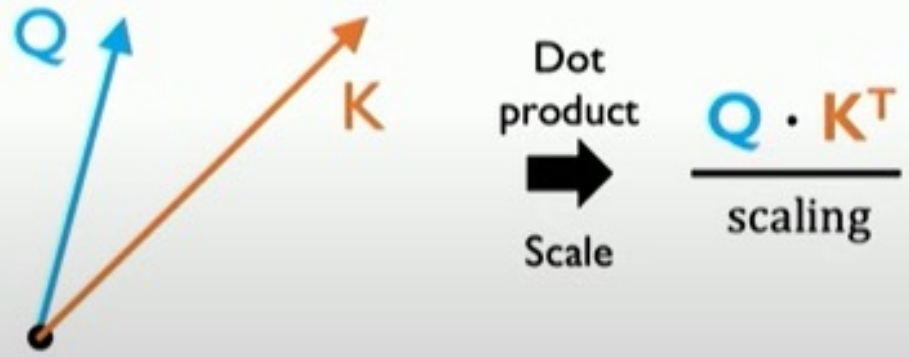
Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

Attention score: compute pairwise similarity between each query and key

How to compute similarity between two sets of features?



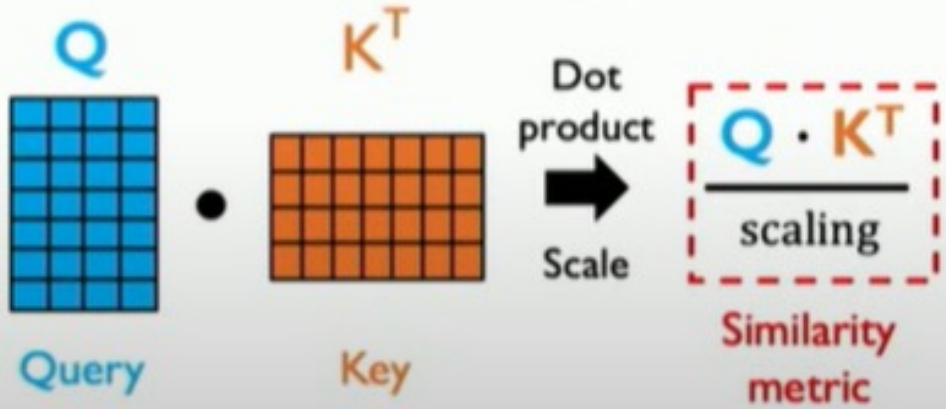
Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

Attention score: compute pairwise similarity between each query and key

How to compute similarity between two sets of features?



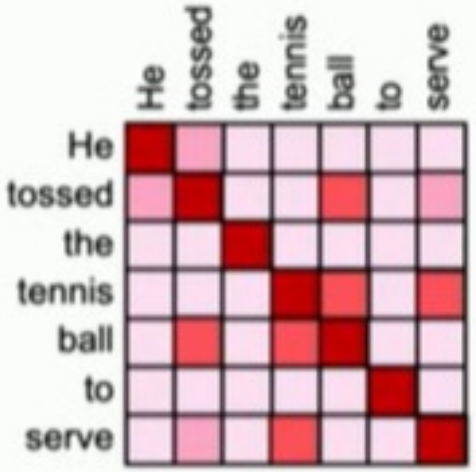
Also known as the "cosine similarity"

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

Attention weighting: where to attend to! How similar is the key to the query?



$$softmax\left(\frac{Q \cdot K^T}{scaling}\right)$$

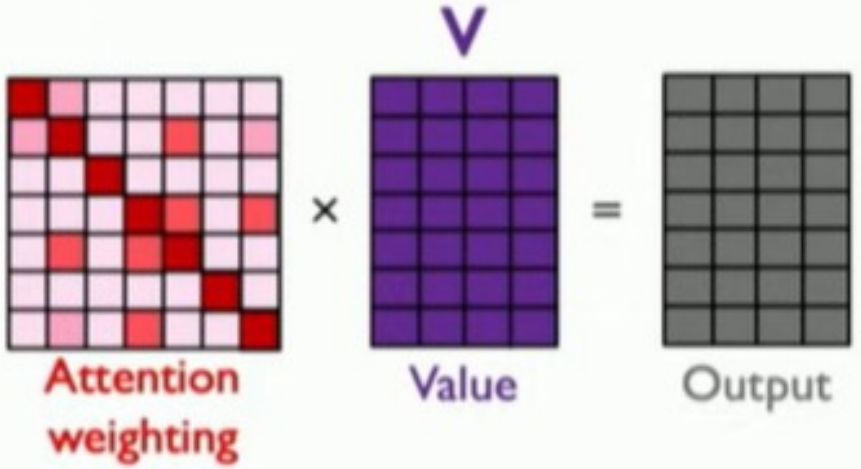
Attention weighting

Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

Last step: self-attend to extract features



$$softmax\left(\frac{Q \cdot K^T}{scaling}\right) \cdot V = A(Q, K, V)$$

Applying Multiple Self-Attention Heads

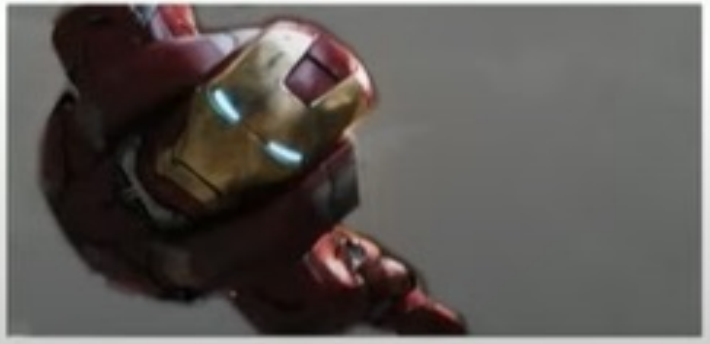
Attention weighting

×

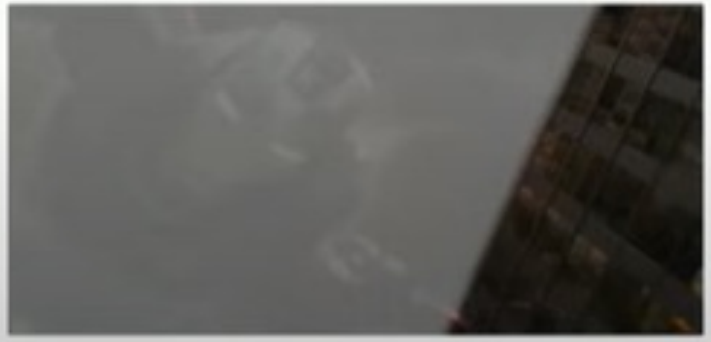
Value

=

Output



Output of attention head 1



Output of attention head 2

Output of attention head 3

Multi-Head Attention

- The encoder's Multi-Head Attention layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones.
  - The output of this layer for the word "Queen" in the sentence "They welcomed the Queen of the United Kingdom" will depend on all the words in the sentence, but it will probably pay more attention to the words "United" and "Kingdom" than to the words "They" or "welcomed"
  - This attention mechanism is called self-attention (the sentence is paying attention to itself)
- The decoder's Masked Multi-Head Attention layer does the same thing, but each word is only allowed to attend to words located before it.
- Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence.
  - The decoder will probably pay close attention to the word "Queen" in the input sentence when it is about to output this word's translation

Visual Attention

- Attention mechanisms are now used for a variety of purposes.
- One of their first applications beyond NMT was in generating image captions using visual attention:
  - A convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time
  - At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image
- In Figure 16-7, the model generated the caption “A woman is throwing a frisbee in a park,” and you can see what part of the input image the decoder focused its attention on when it was about to output the word “frisbee”
  - Most of its attention was focused on the frisbee.

Figure 16-7. Visual attention: an input image (left) and the model’s focus before producing the word “frisbee” (right)[18]

Explainability

- Explainability is one extra benefit of attention mechanisms. They make it easier to understand what led the model to produce its output.

- It can be especially useful when the model makes a mistake:

- An example from a 2016 paper by Marco Tulio Ribeiro et al.:
  - If an image of a dog walking in the snow is labeled as “a wolf walking in the snow,” then you can go back and check what the model focused on when it output the word “wolf”
  - You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there’s a lot of snow around.
  - You can then fix this by training the model with more images of wolves without snow, and dogs with snow

Recent Innovations in Language Models

- The **GPT** (Generative Pre-trained Transformer) paper by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining, but this time using a Transformer-like architecture.
- The authors pretrained a large but fairly simple architecture composed of a **stack of 12 Transformer modules** (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self- supervised learning.
- Then they **fine-tuned it on various language tasks**, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, entailment (whether sentence A entails sentence B), similarity (e.g., “Nice weather today” is very similar to “It is sunny”), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions).
- Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the **GPT-2** paper, which proposed a very similar architecture, but larger still (***with over 1.5 billion parameters***!) and they showed that it could achieve good performance on many tasks without any fine-tuning. This is called zero-shot learning (ZSL).
- A smaller version of the GPT-2 model (with “just” **117 million parameters**) is available at https://github.com/openai/gpt-2, along with its pretrained weights.

BERT - Bidirectional Encoder Representations from Transformers

- The BERT paper by Jacob Devlin and other Google researchers also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer's encoder).

- This means that the model is naturally bidirectional; hence the B in BERT (Bidirectional Encoder Representations from Transformers).

- Most importantly, the authors proposed two pretraining tasks that explain most of the model's strength

Masked Language Model (MLM)

- Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words.

- For example, if the original sentence is “She had fun at the birthday party,” then the model may be given the sentence “She <mask> fun at the <mask> party” and it must predict the words “had” and “birthday” (the other outputs will be ignored).

- To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

Main Innovations

- The main innovations in 2018 and 2019 have been:
  - Better subword **tokenization**
  - Shifting from LSTMs to **Transformers**
  - Pretraining universal language models using **self-supervised learning**, then fine-tuning them with very few architectural changes (or none at all).

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Representation Learning and Generative Learning using Autoencoders and GANs

Professor Hamza F. Alsarhan
SEAS 8505

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

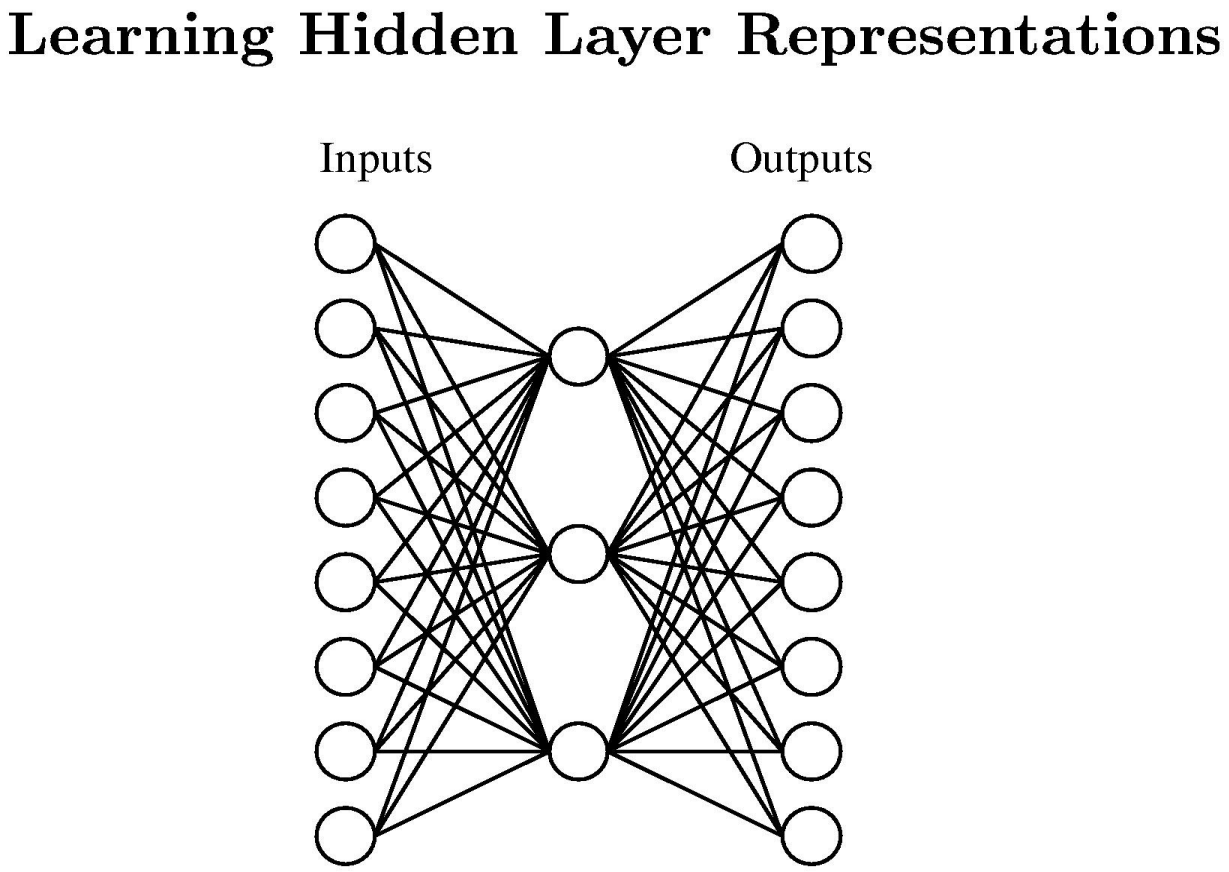
Autoencoders

Professor Hamza F. Alsarhan
SEAS 8505

Overview

- Autoencoders are artificial neural networks capable of learning *dense representations of the input data*, called **latent representations** or codings, without any supervision (i.e., the training set is unlabeled)
  - These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 8), especially for visualization purposes.
- Autoencoders also act as **feature detectors**, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in Chapter 11).
- Some autoencoders are generative models:
  - Capable of randomly generating new data that looks very similar to the training data
- Autoencoders and Generative Adversarial Networks (GANs) are both unsupervised, they both learn dense representations, they can both be used as generative models, and they have many similar applications.
  - However, they work very differently

Autoencoders



Based on Domingos

Autoencoders

A target function:

Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned?

Based on Domingos

Autoencoders

Learned hidden layer representation:

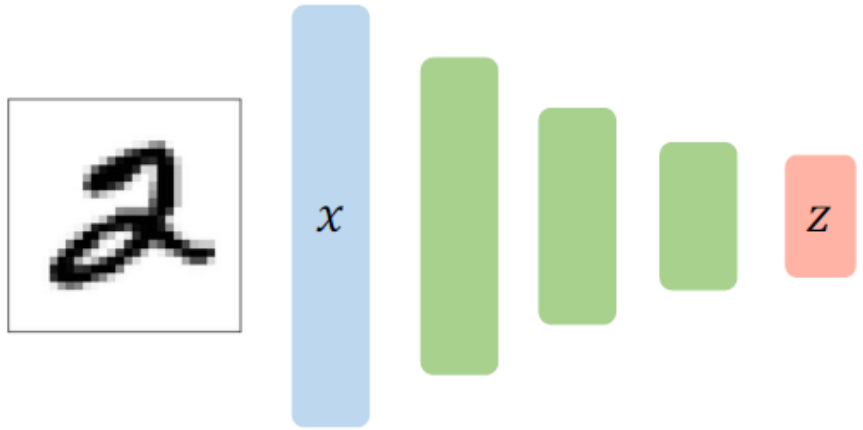
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Based on Domingos

Autoencoders

Autoencoders: background

Unsupervised approach for learning a **lower-dimensional** feature representation from unlabeled training data



Why do we care about a low-dimensional z?

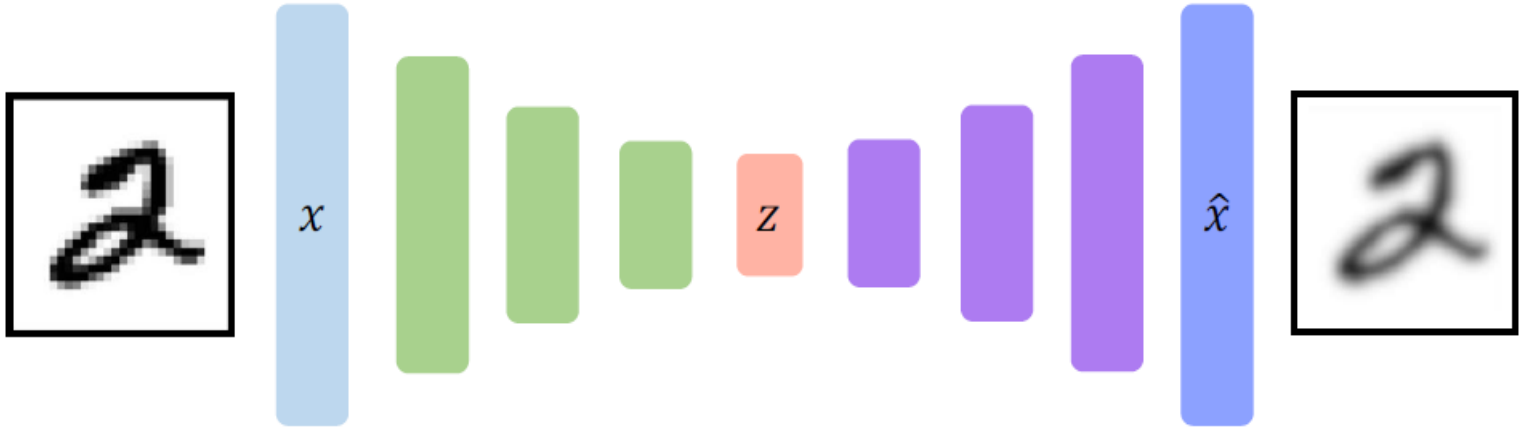
"Encoder" learns mapping from the data, x, to a low-dimensional latent space, z

http://introtodeeplearning.com/

Autoencoders

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**



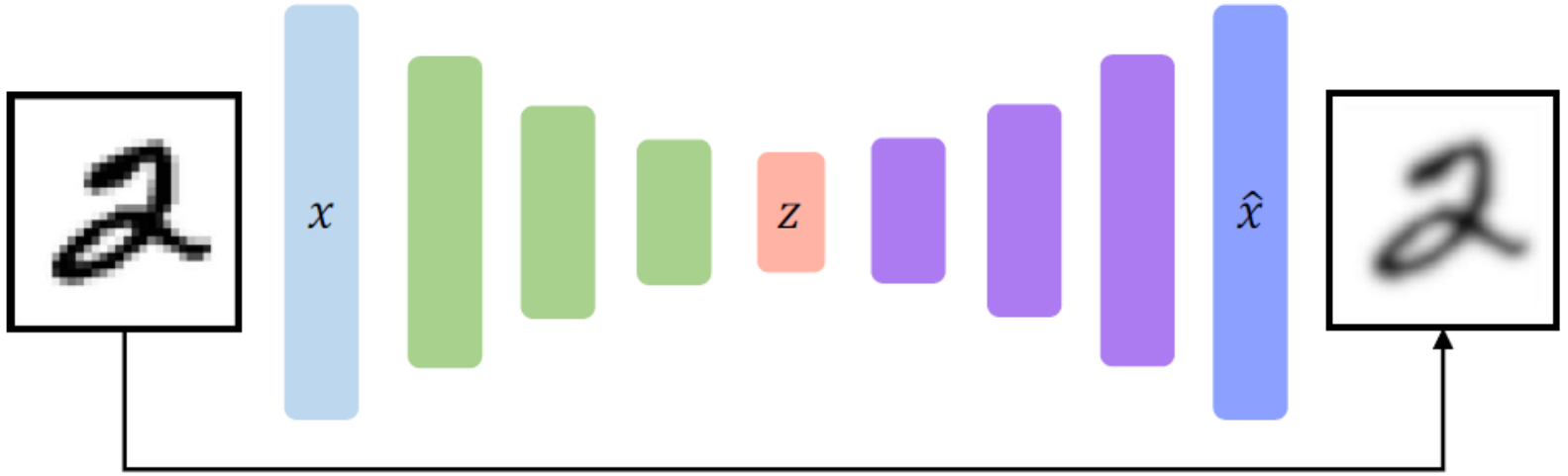
"Decoder" learns mapping back from latent, z, to a reconstructed observation, $\hat{x}$

http://introtodeeplearning.com/

Autoencoders

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**



$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

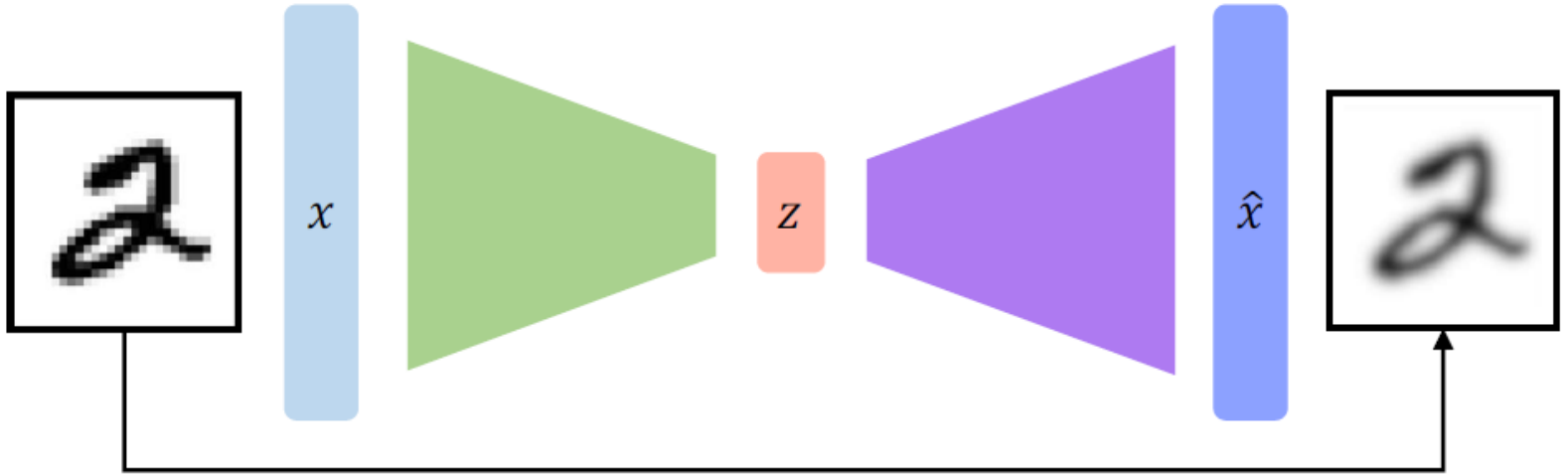
Loss function doesn't use any labels!!

http://introtodeeplearning.com/

Autoencoders

Autoencoders: background

How can we learn this latent space?
Train the model to use these features to **reconstruct the original data**



$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

Loss function doesn't use any labels!!

http://introtodeeplearning.com/

Dimensionality of latent space → reconstruction quality

Autoencoding is a form of compression!
Smaller latent space will force a larger training bottleneck

2D latent space

5D latent space

Ground Truth

http://introtodeeplearning.com/

Autoencoders for representation learning

Bottleneck hidden layer forces network to learn a compressed latent representation

Reconstruction loss forces the latent representation to capture (or encode) as much "information" about the data as possible

Autoencoding = **Auto**matically **encoding** data

http://introtodeeplearning.com/

Efficient Data Representations

- An autoencoder looks at the inputs, converts them to an efficient latent representation, and then spits out something that looks very close to the inputs
- An autoencoder is always composed of two parts:
  - An encoder (or recognition network) that converts the inputs to a latent representation
  - Followed by a decoder (or generative network) that converts the internal representation to the outputs
- You can think of autoencoders as a form of self-supervised learning (i.e., using a supervised learning technique with automatically generated labels, in this case simply equal to the inputs)

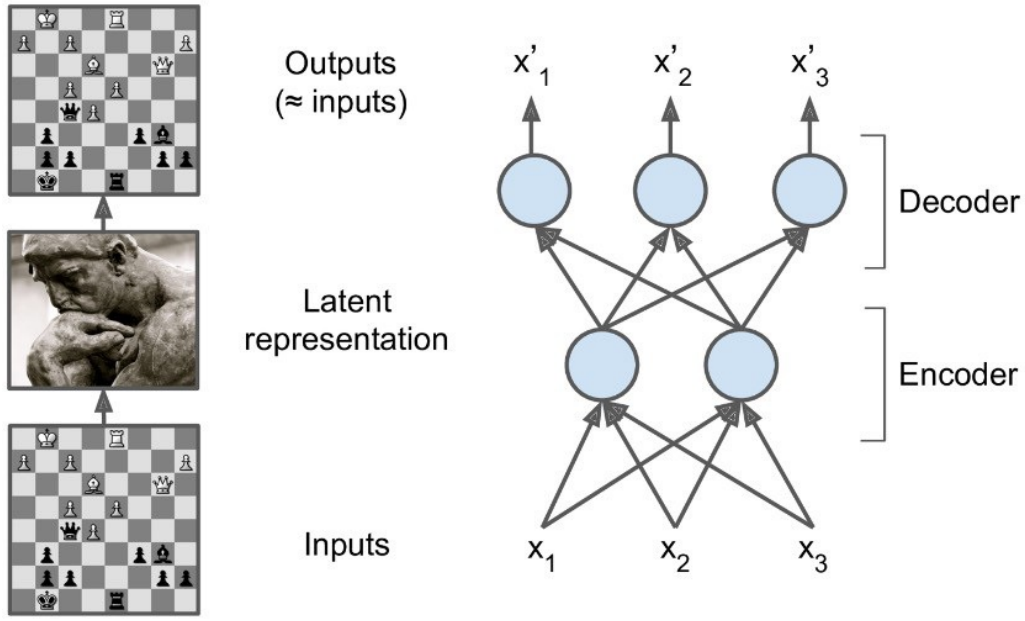


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

Learns the Most Important Features

- An autoencoder typically has the same architecture as a Multi-Layer Perceptron, except that the number of neurons in the output layer must be equal to the number of inputs
- In this example on the right, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder)
- The autoencoder tries to reconstruct the inputs, and the cost function contains a reconstruction loss that penalizes the model when the reconstructions are different from the inputs
- Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be undercomplete.
  - An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs
  - **Forced to learn the most important features in the input data (and drop the unimportant ones)**

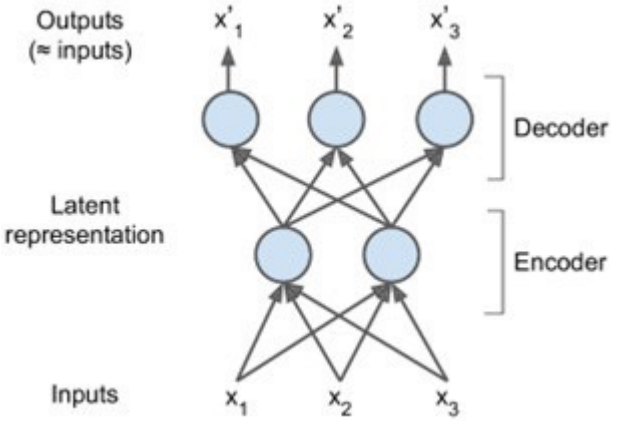


Figure 17-4. Original images (top) and their reconstructions (bottom)

Performing PCA with an Undercomplete Linear Autoencoder

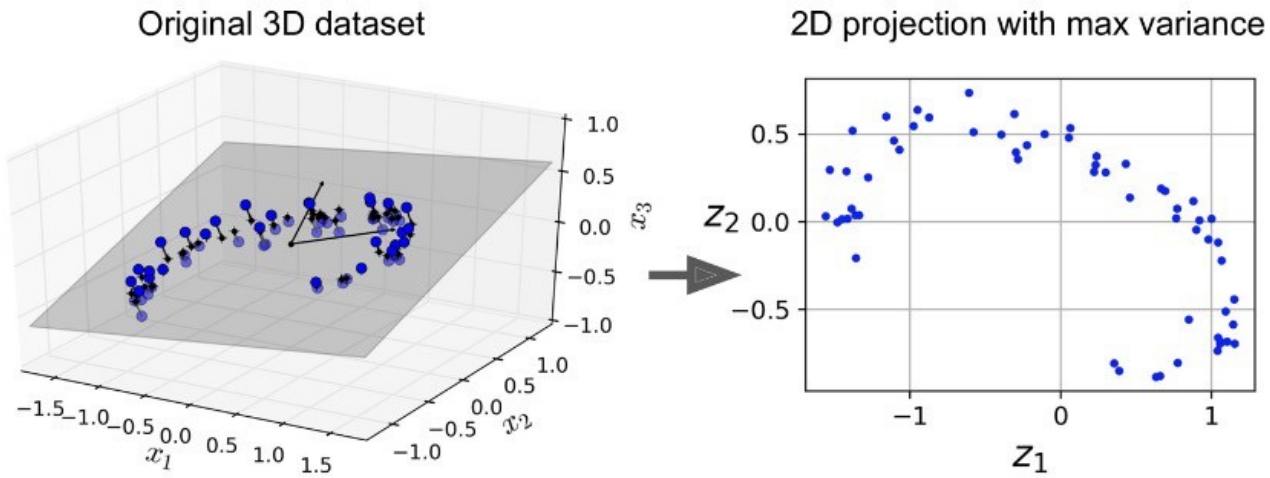


Figure 17-2. PCA performed by an undercomplete linear autoencoder

- If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing Principal Component Analysis

Stacked Autoencoders

- The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer)
- Adding more layers helps the autoencoder learn more complex codings

- *Be careful not to make the autoencoder too powerful*
  - A powerful encoder will just learn to map each input to a single arbitrary number (and the decoder learns the reverse mapping)
  - Such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances)

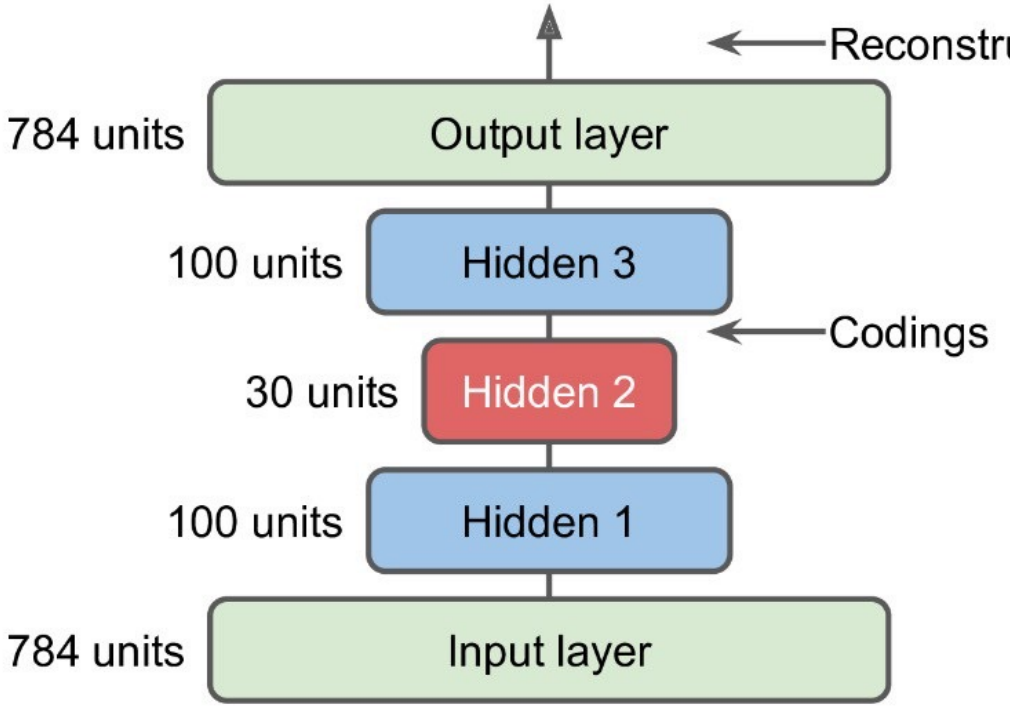


Figure 17-3. Stacked autoencoder

Visualizing the Fashion MNIST Dataset

- You can use a trained stacked autoencoder to reduce a dataset's dimensionality
- For visualization, this does not give great results compared to other dimensionality reduction algorithms.
  - But one big advantage of autoencoders is that they can handle large datasets, with many instances and many features.
- For Fashion MNIST:
  - First, we use the encoder from our stacked autoencoder to reduce the dimensionality down to 30
  - then use Scikit-Learn's implementation of the t-distributed Stochastic Neighbor Embedding (t-SNE) algorithm to reduce the dimensionality down to 2 for visualization

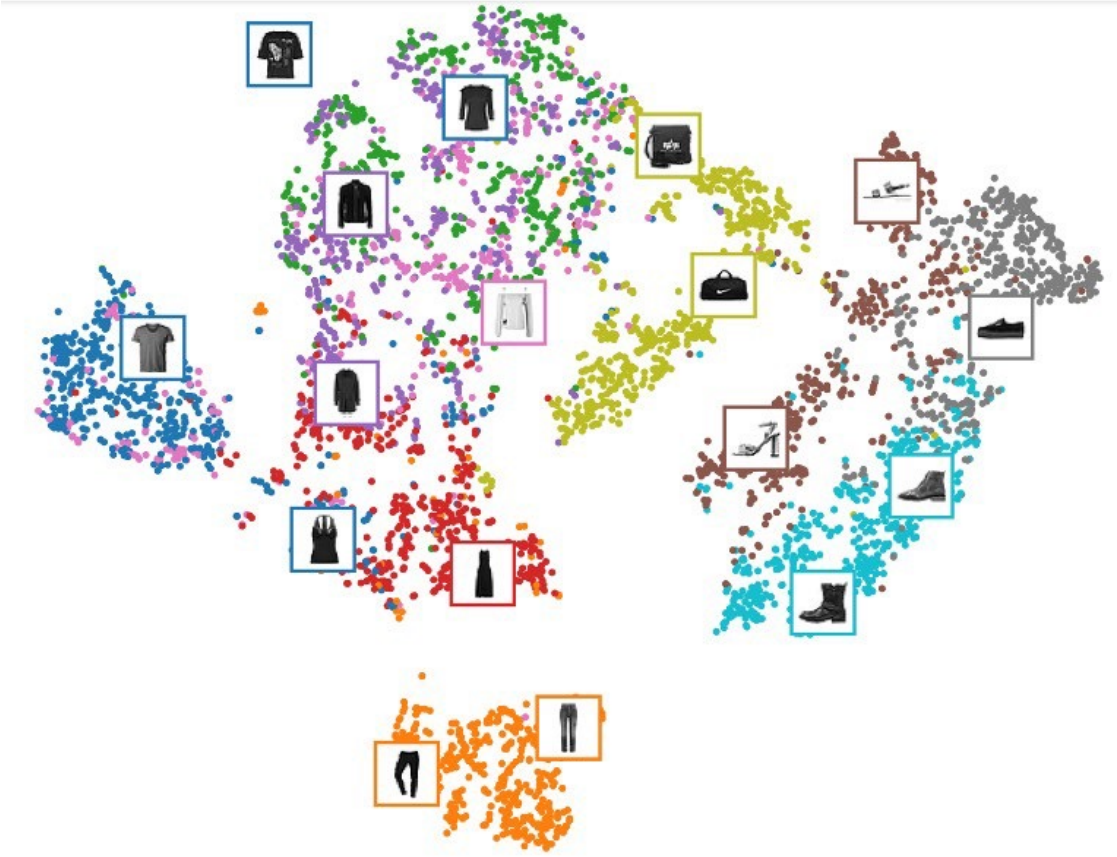


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

Figure 17-4. Original images (top) and their reconstructions (bottom)

Unsupervised Pretraining Using Stacked Autoencoders

- If you have a large dataset but most of the data is unlabeled
  - First train a stacked autoencoder using all the data
  - Then ***reuse the lower layers*** to create a neural network for your actual task
  - Train the classifier using the labeled data
- When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones)

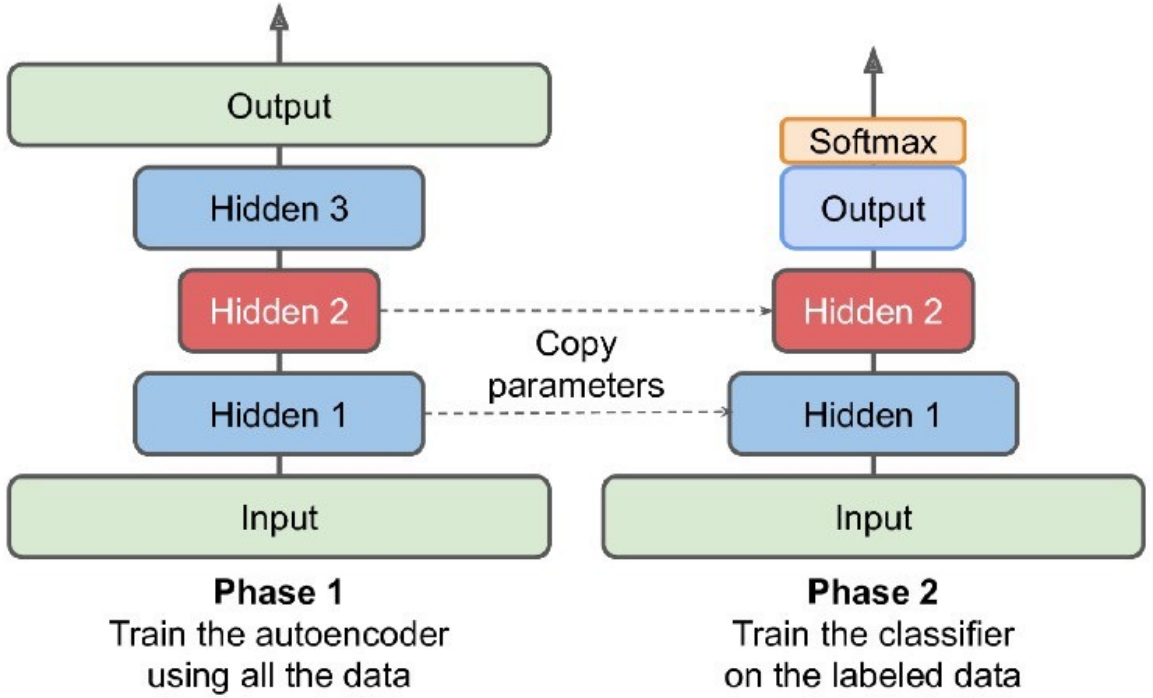


Figure 17-6. Unsupervised pretraining using autoencoders

Motivation for Pretraining with Autoencoders

- Having plenty of unlabeled data and little labeled data is common.
- Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans.
- Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances.

Convolutional Autoencoders

- If you want to build an autoencoder for ***images***, you will need to build a convolutional autoencoder (unless the images are very small)
  - Convolutional neural networks are far better suited than dense networks to work with images
- The encoder is a regular CNN composed of convolutional layers and pooling layers
  - It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps)
- The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions)
  - Can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers)

Recurrent Autoencoders

- If you want to build an autoencoder for ***sequences***, such as time series or text (e.g., for unsupervised learning or dimensionality reduction), then recurrent neural networks may be better suited than dense networks.

- Building a recurrent autoencoder is straightforward:
  - The encoder is typically a sequence-to-vector RNN which compresses the input sequence down to a single vector
  - The decoder is a vector-to-sequence RNN that does the reverse

Denoising Autoencoders

- Adding noise to its inputs is another way to force the autoencoder to learn useful features
  - Trains it to recover the original, noise-free inputs
- The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout

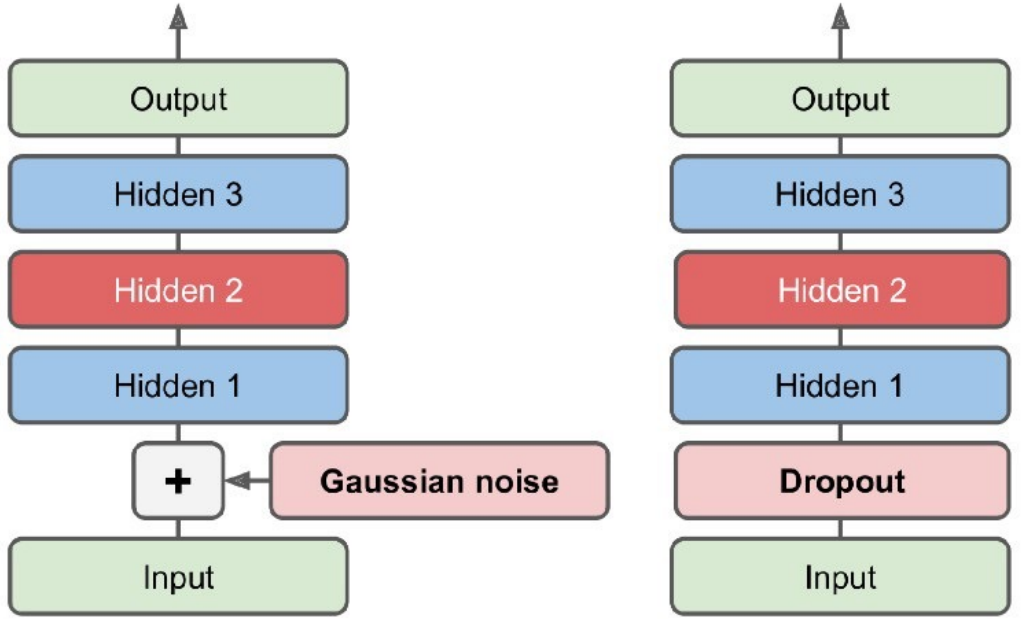


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

- Another kind of constraint that often leads to good feature extraction is sparsity:
  - By adding an appropriate term to the cost function, the autoencoder is pushed to ***reduce the number of active neurons*** in the coding layer

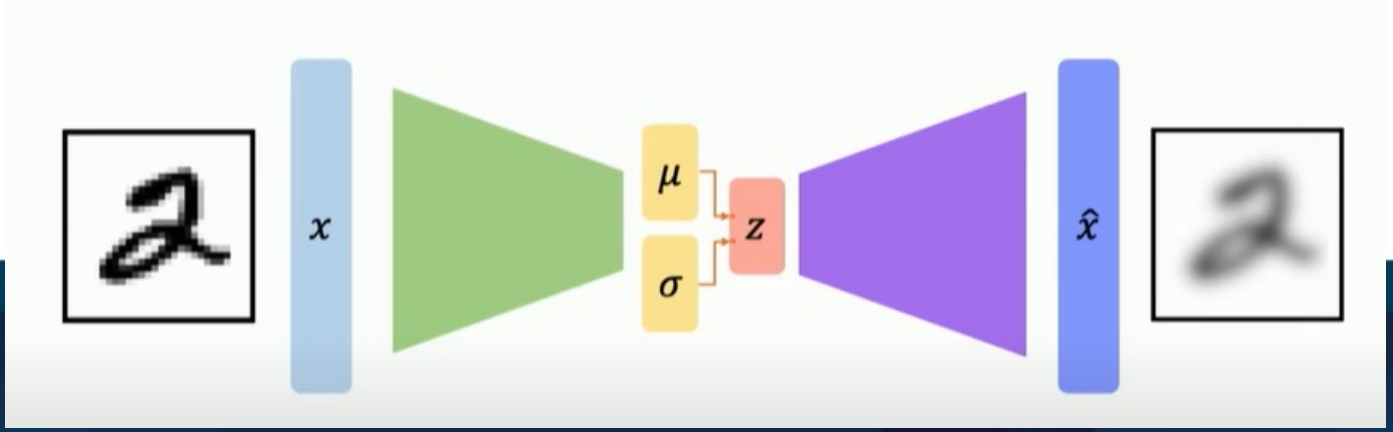
- This forces the autoencoder to represent each input as a combination of a small number of activations

- As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to)

Variational Autoencoders

- Variational autoencoders were introduced in 2013 and quickly became one of the most popular types of autoencoders

- Different from other autoencoders in these particular ways:
  - Probabilistic – their outputs are partly determined by chance, even after training
    - As opposed to denoising autoencoders, which use randomness only during training)
  - Generative – they can generate new instances that look like they were sampled from the training set



Variational Autoencoders – How They Work

- Follows the basic structure of all autoencoders with an encoder followed by a decoder but injects randomness into the encoder
- A variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution during training, the cost function pushes the codings to gradually migrate within the coding space (also called the latent space) to end up looking like a cloud of Gaussian points.
- One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

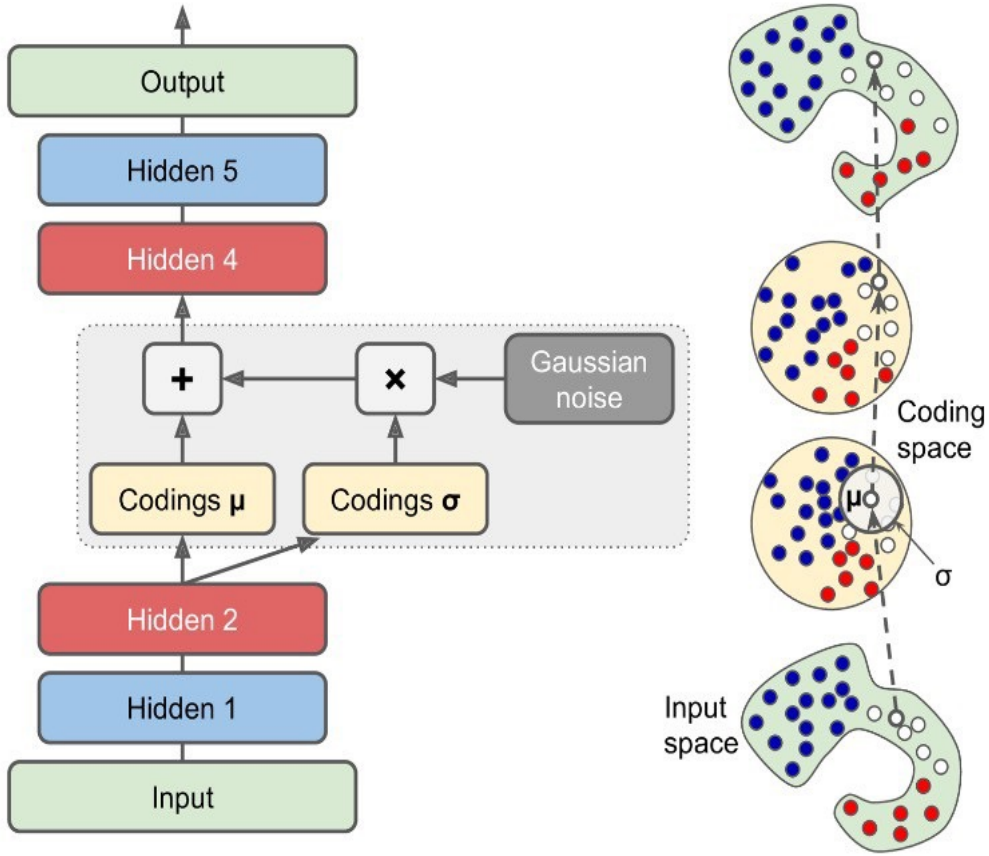


Figure 17-12. Variational autoencoder (left) and an instance going through it (right)

Cost Function of Variational Autoencoders

- Composed of two parts
  - The first is the usual **reconstruction loss** that pushes the autoencoder to reproduce its inputs (we can use the MSE for this).
  - The second is the ***latent loss*** that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution. It is the divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings.

Intermediate Images

- Now let's use a variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them.
- Figure 17-12 shows the generated images.
- The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn!
- Variational autoencoders make it possible to perform *semantic interpolation:* instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level.
- For example, let's take a few codings along an arbitrary line in latent space and decode them. We get a sequence of images that gradually go from pants to sweaters

Figure 17-12. Fashion MNIST images generated by the variational autoencoder

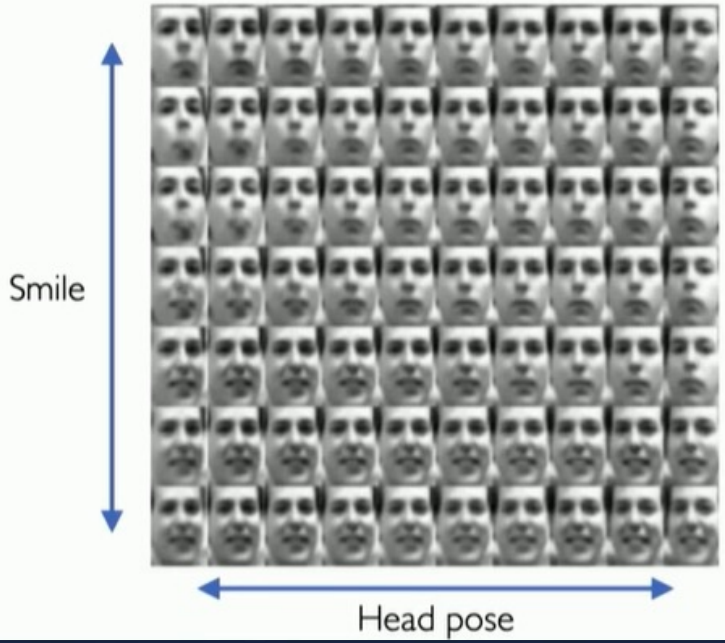
Figure 17-13. Semantic interpolation

Other things we can do with VAEs

VAEs: Latent perturbation

Slowly increase or decrease a **single latent variable**
Keep all other variables fixed

VAEs: Latent perturbation



Ideally, we want latent variables that are uncorrelated with each other

Enforce diagonal prior on the latent variables to encourage independence

Disentanglement

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Generative Adversarial Networks (GANs)

Professor Hamza F. Alsarhan
SEAS 8505

Challenge: Which is the Real Face?

Generative Adversarial Networks (GANs)

- GANs are composed of two neural networks:
  - A **generator** that tries to generate data that looks similar to the training data
  - A **discriminator** that tries to tell real data from fake data
- This architecture is very original in Deep Learning in that the generator and the discriminator compete against each other during training:
  - A way to think about it - the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake
- Adversarial training (training competing neural networks) is widely considered as one of the most important ideas in recent years

Uses for GANs

- GANs are now widely used for:
  - super resolution (increasing the resolution of an image)
  - colorization
  - powerful image editing (e.g., replacing photo sections with a realistic background)
  - turning a simple sketch into a photorealistic image
  - predicting the next frames in a video
  - augmenting a dataset (to train other models)
  - generating other types of data (such as text, audio, and time series)
  - identifying the weaknesses in other models and strengthening them

Supervised vs unsupervised learning

Supervised Learning

Data: (x, y)
x is data, y is label

Goal: Learn function to map
$x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, etc.

Unsupervised Learning

Data: x
x is data, no labels!

Goal: Learn some *hidden* or *underlying structure* of the data

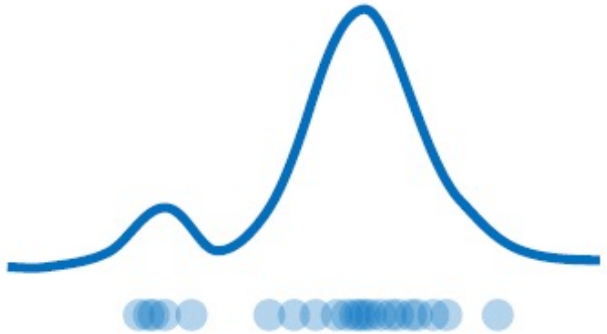
Examples: Clustering, feature or dimensionality reduction, etc.

http://introtodeeplearning.com/

Generative modeling

Goal: Take as input training samples from some distribution and learn a model that represents that distribution

Density Estimation



Sample Generation

Input samples

Training data ~ $P_{data}(x)$

Generated samples

Generated ~ $P_{model}(x)$

How can we learn $P_{model}(x)$ similar to $P_{data}(x)$?

http://introtodeeplearning.com/

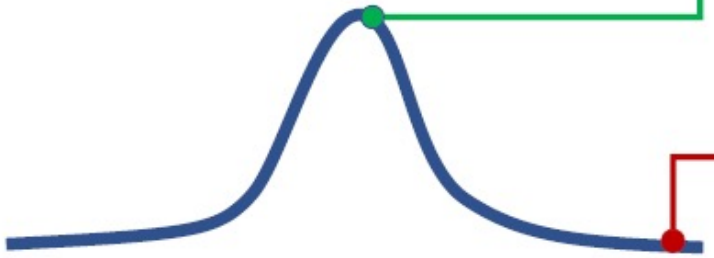
Why generative models? Outlier detection

- **Problem:** How can we detect when we encounter something new or rare?
- **Strategy:** Leverage generative models, detect outliers in the distribution
- Use outliers during training to improve even more!

95% of Driving Data:

(1) sunny, (2) highway, (3) straight road

Detect outliers to avoid unpredictable behavior when training



Edge Cases

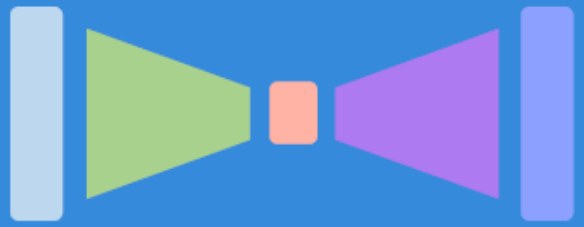
Harsh Weather

Pedestrians

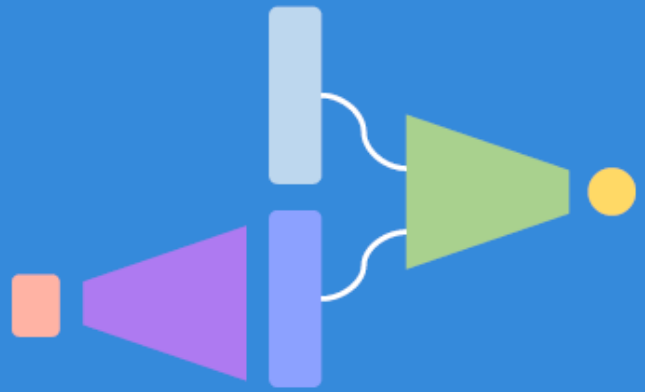
http://introtodeeplearning.com/

Latent variable models

Autoencoders and Variational Autoencoders (VAEs)



Generative Adversarial Networks (GANs)



http://introtodeeplearning.com/

What is a latent variable?

Can we learn the **true explanatory factors**, e.g. latent variables, from only observed data?

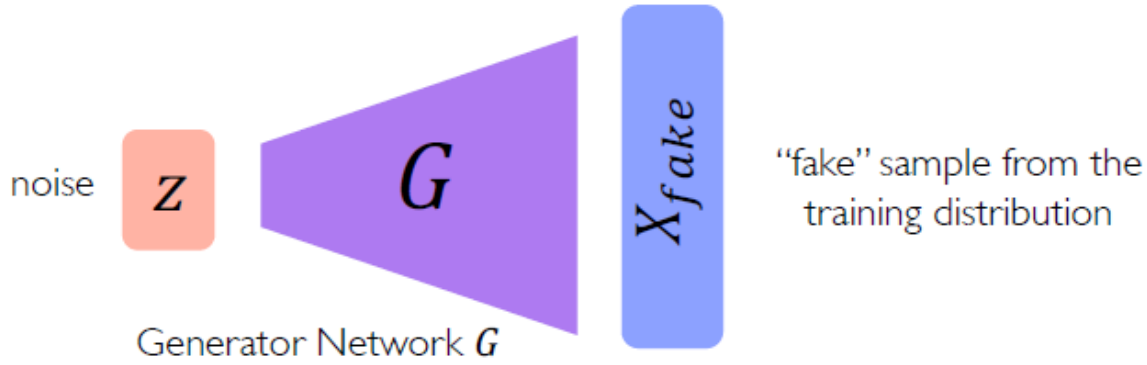
http://introtodeeplearning.com/

What if we just want to sample?

Idea: don't explicitly model density, and instead just sample to generate new instances.

Problem: want to sample from complex distribution – can't do this directly!

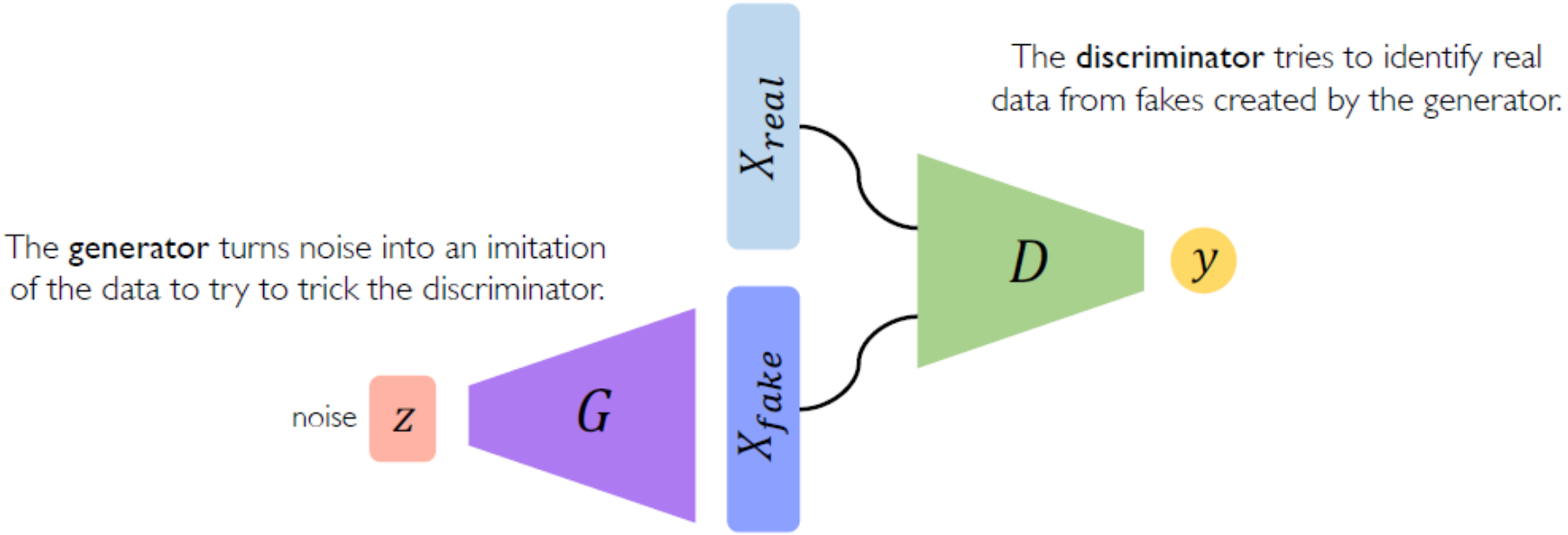
Solution: sample from something simple (noise), learn a transformation to the training distribution.



http://introtodeeplearning.com/

Generative Adversarial Networks (GANs)

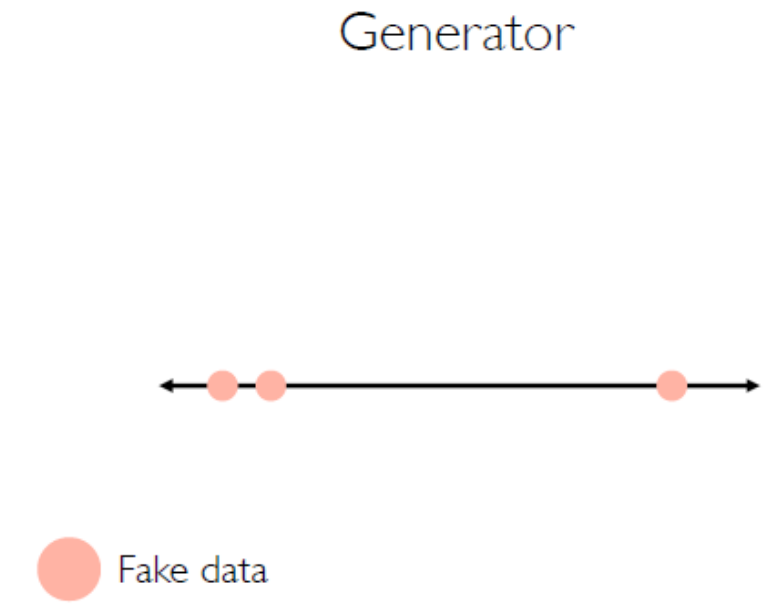
Generative Adversarial Networks (GANs) are a way to make a generative model by having two neural networks compete with each other.



http://introtodeeplearning.com/

Intuition behind GANs

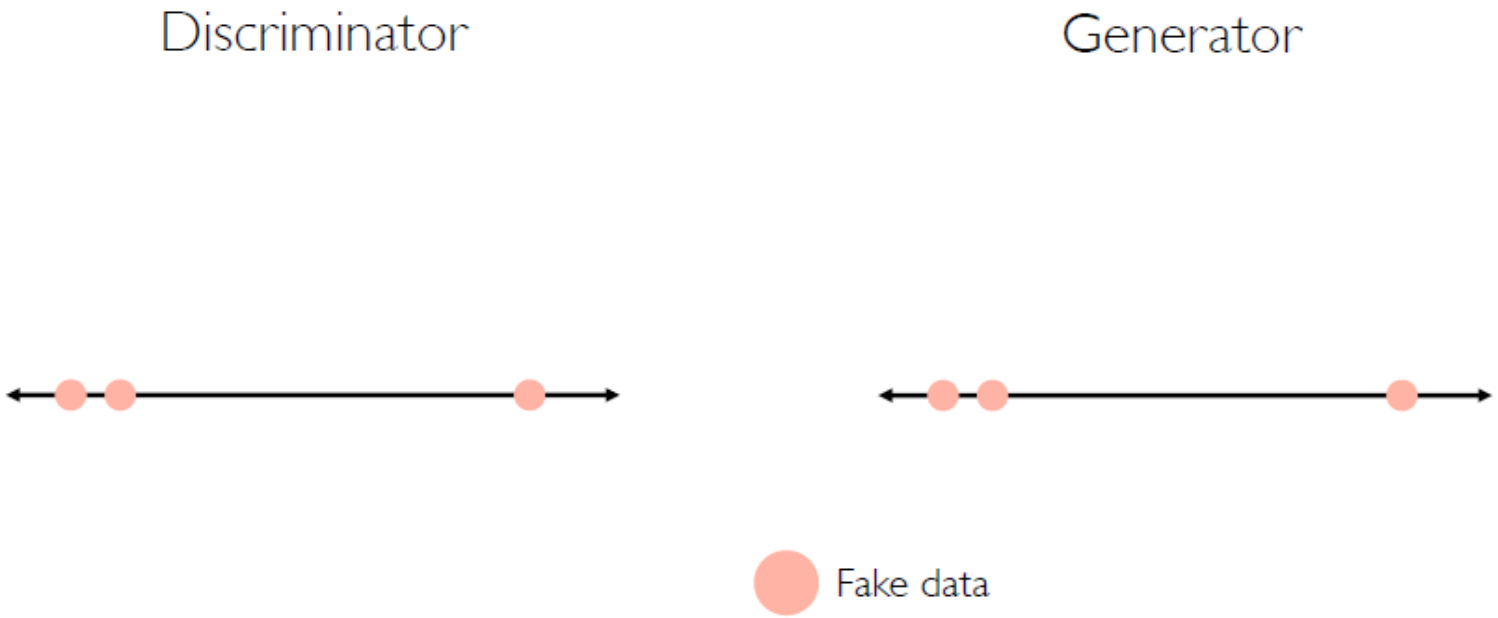
Generator starts from noise to try to create an imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

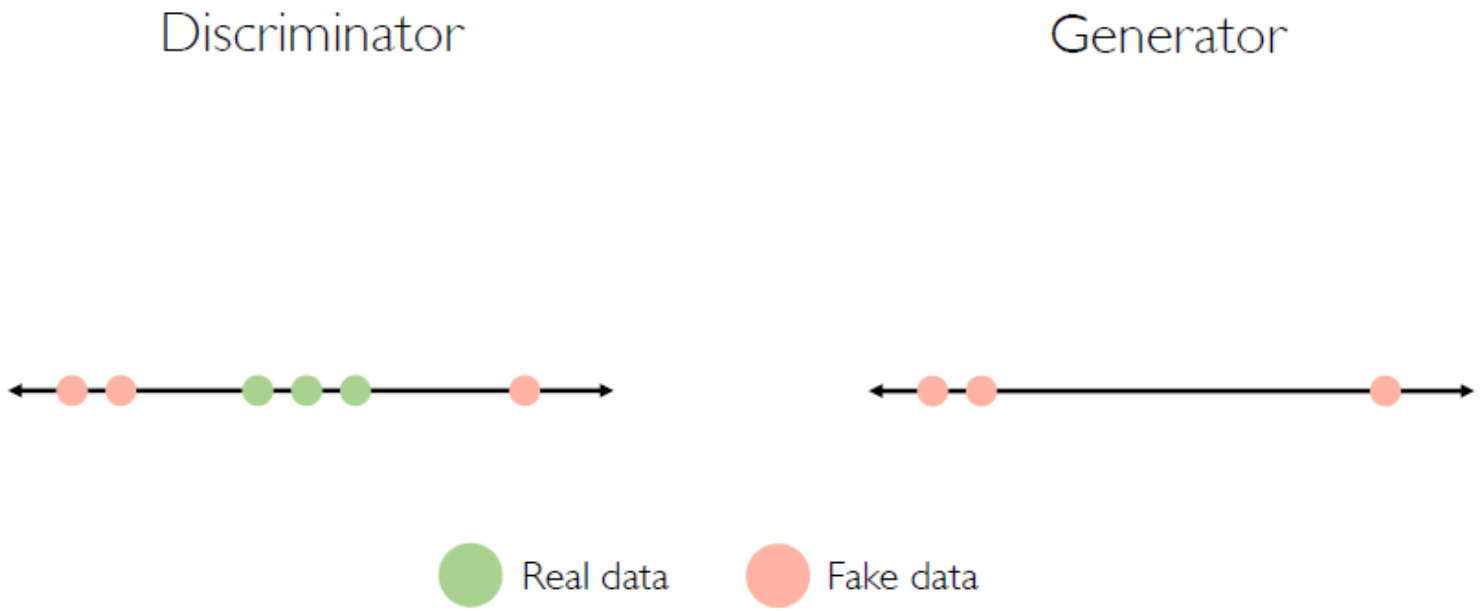
Discriminator looks at both real data and fake data created by the generator.



http://introtodeeplearning.com/

Intuition behind GANs

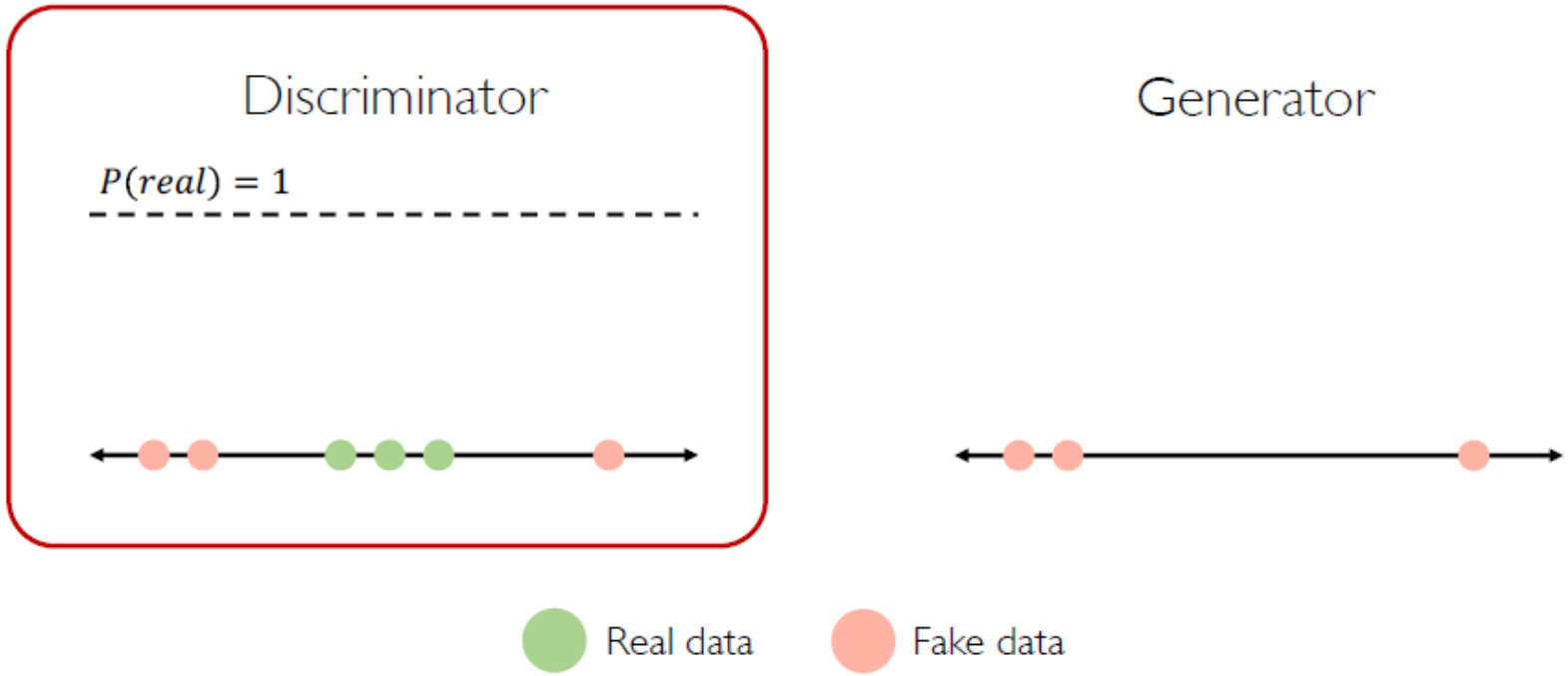
Discriminator looks at both real data and fake data created by the generator.



http://introtodeeplearning.com/

Intuition behind GANs

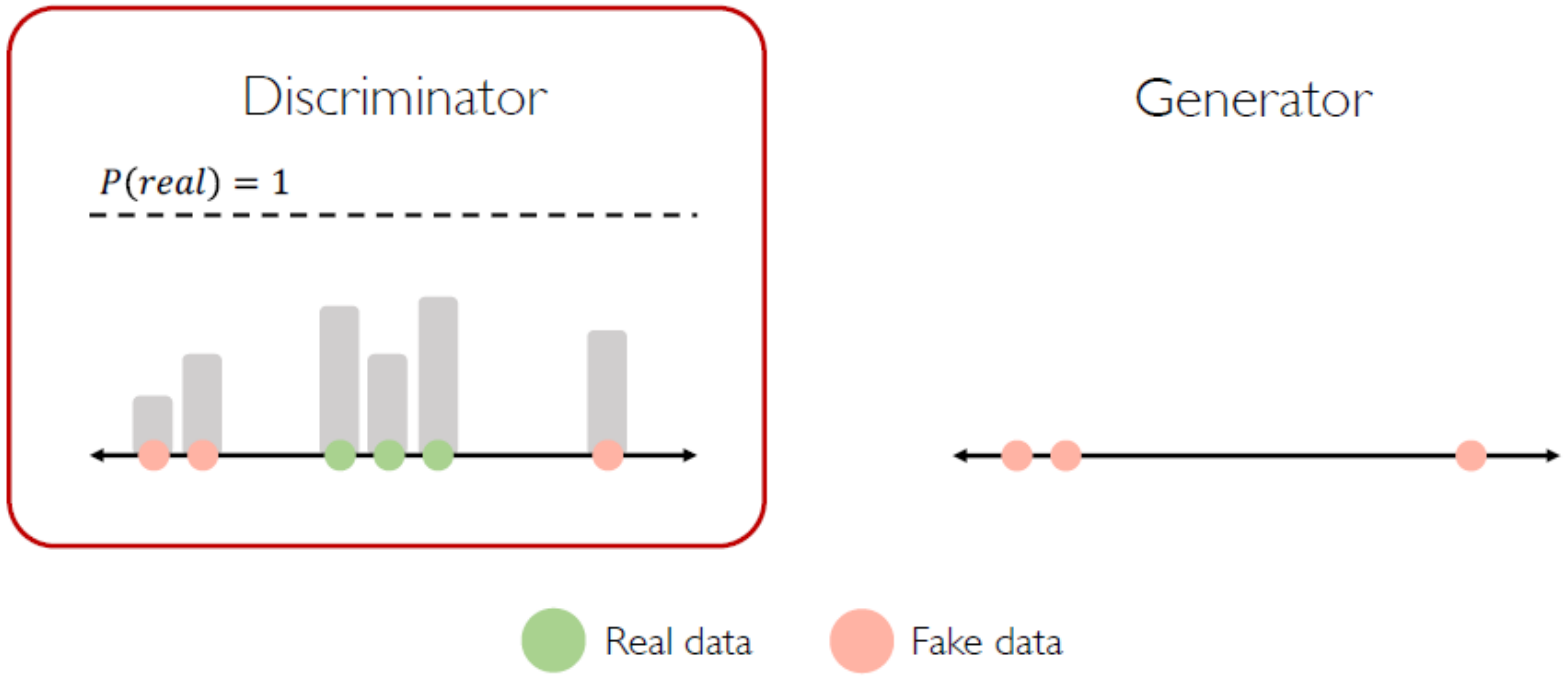
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

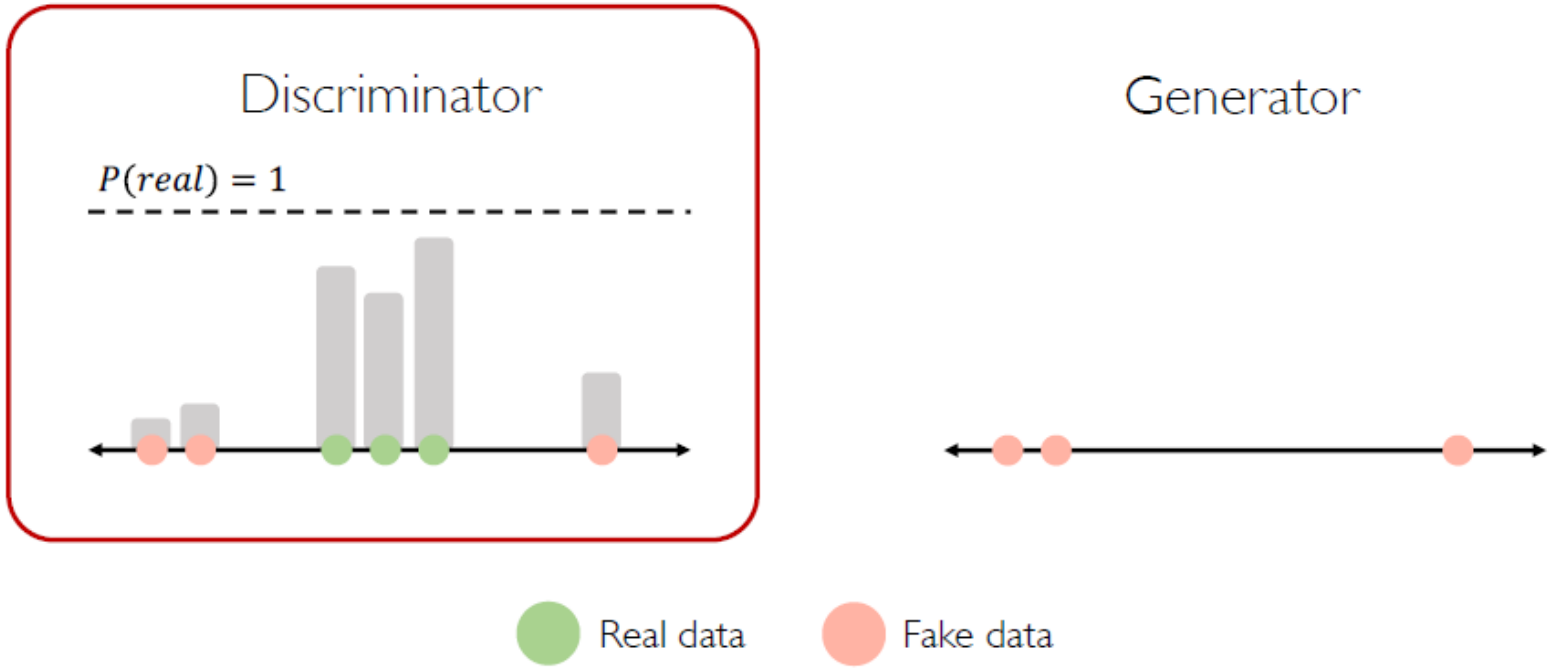
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

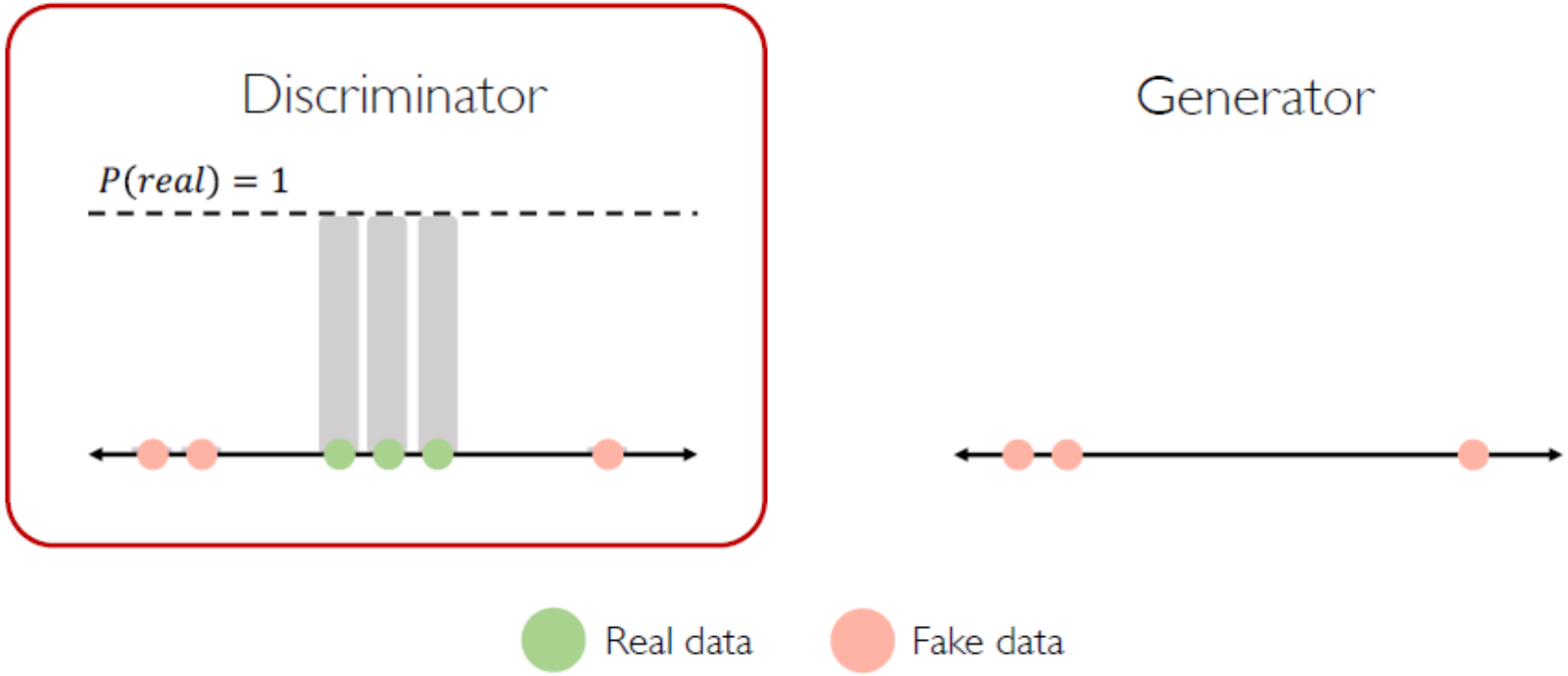
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

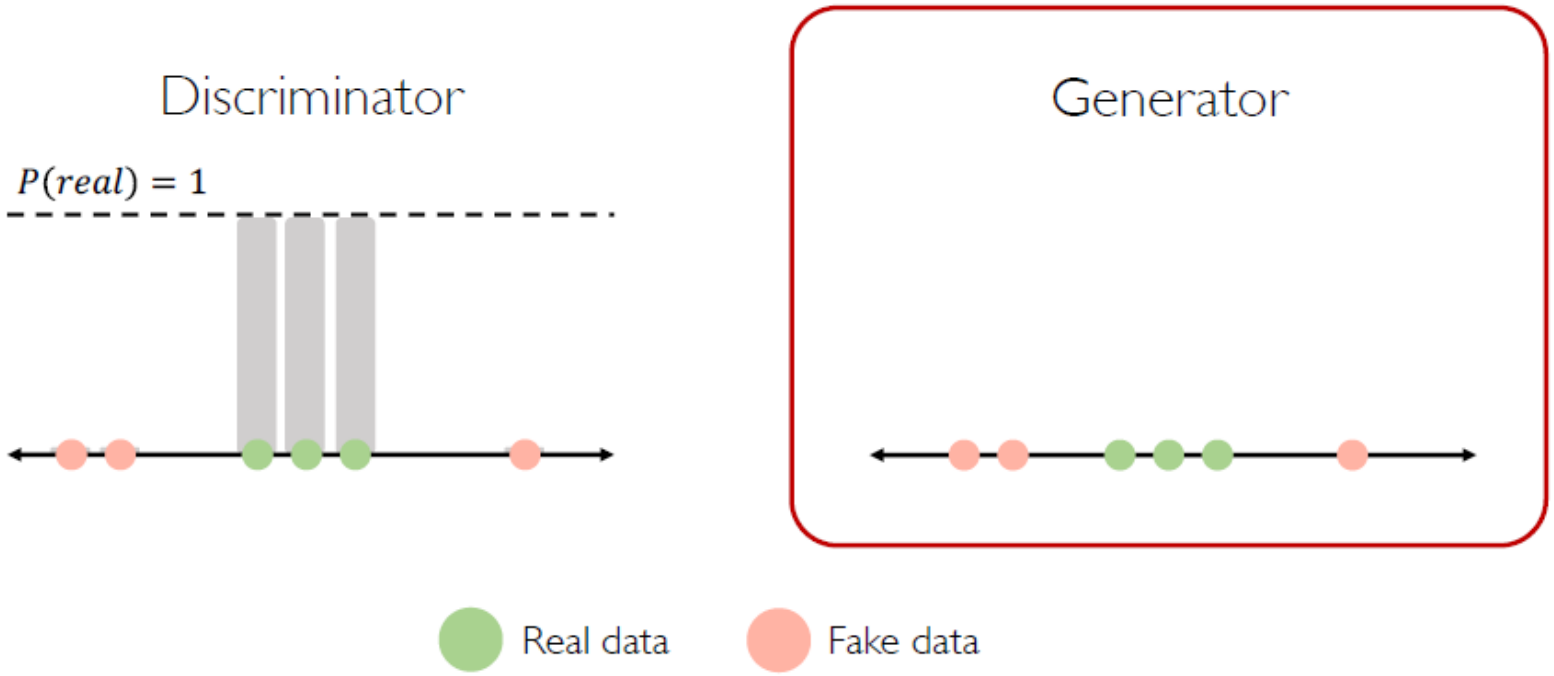
Intuition behind GANs

Generator tries to improve its imitation of the data.

http://introtodeeplearning.com/

Intuition behind GANs

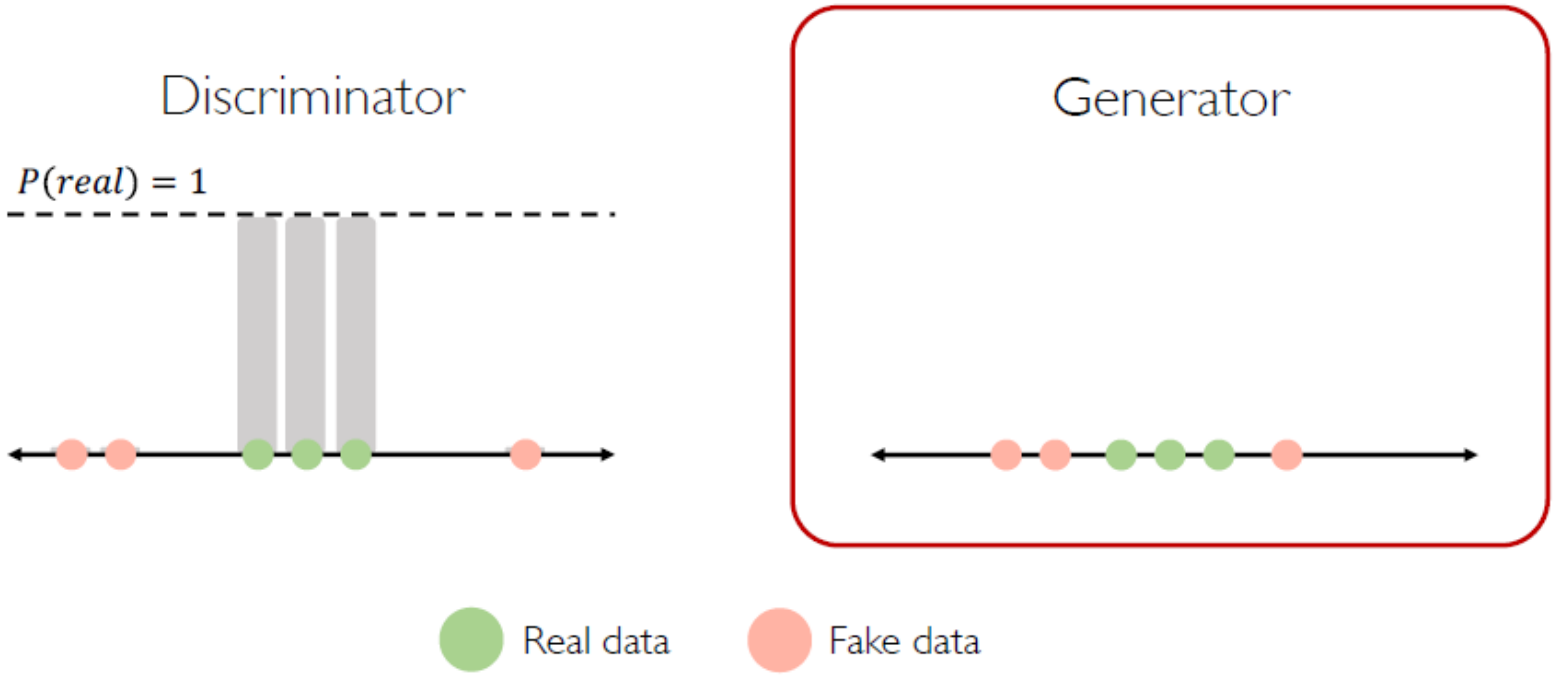
Generator tries to improve its imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

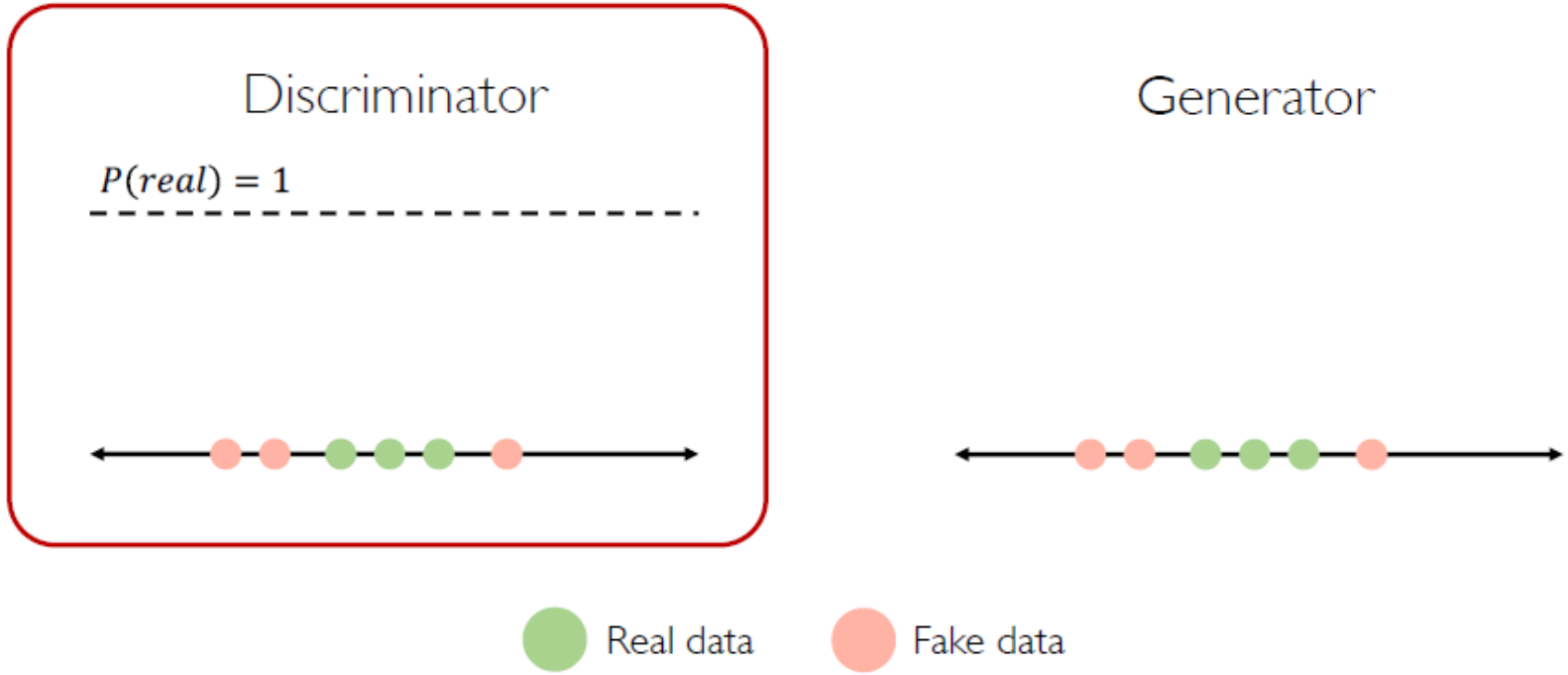
Generator tries to improve its imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

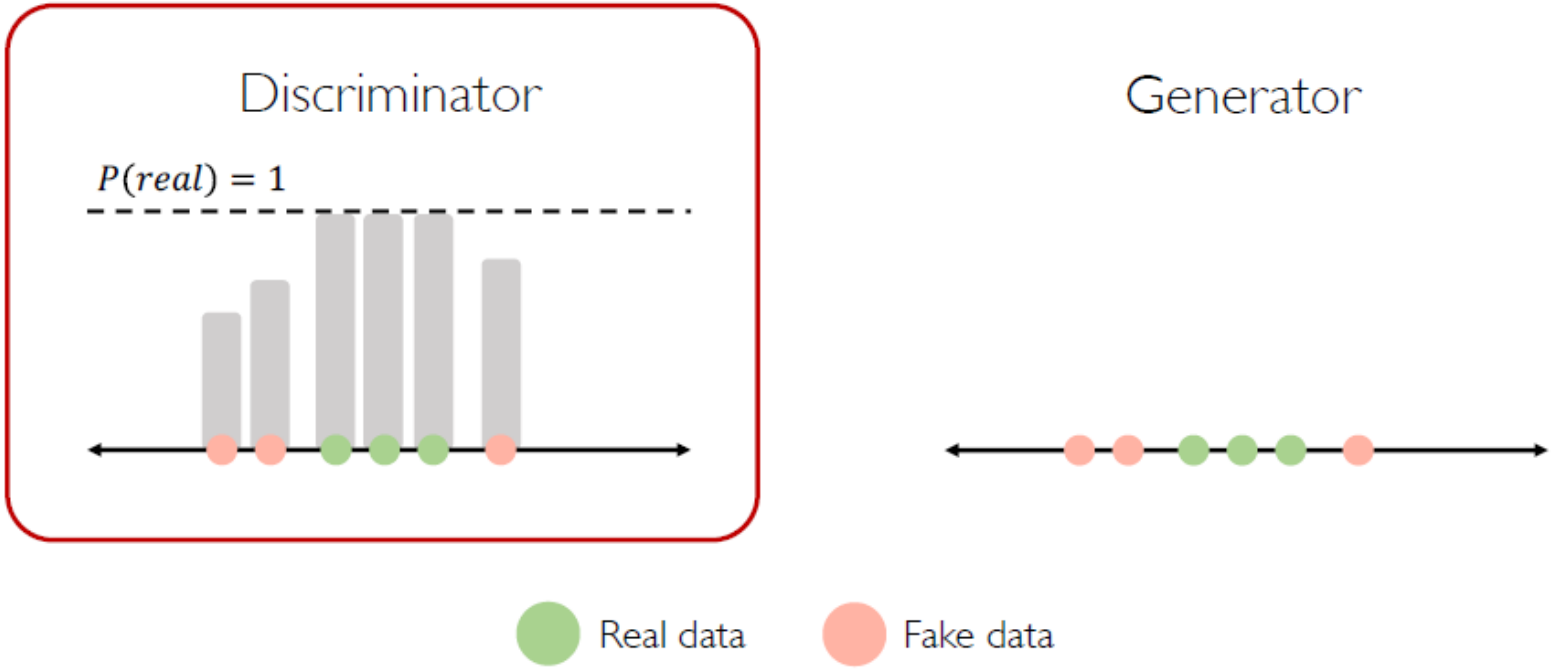
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

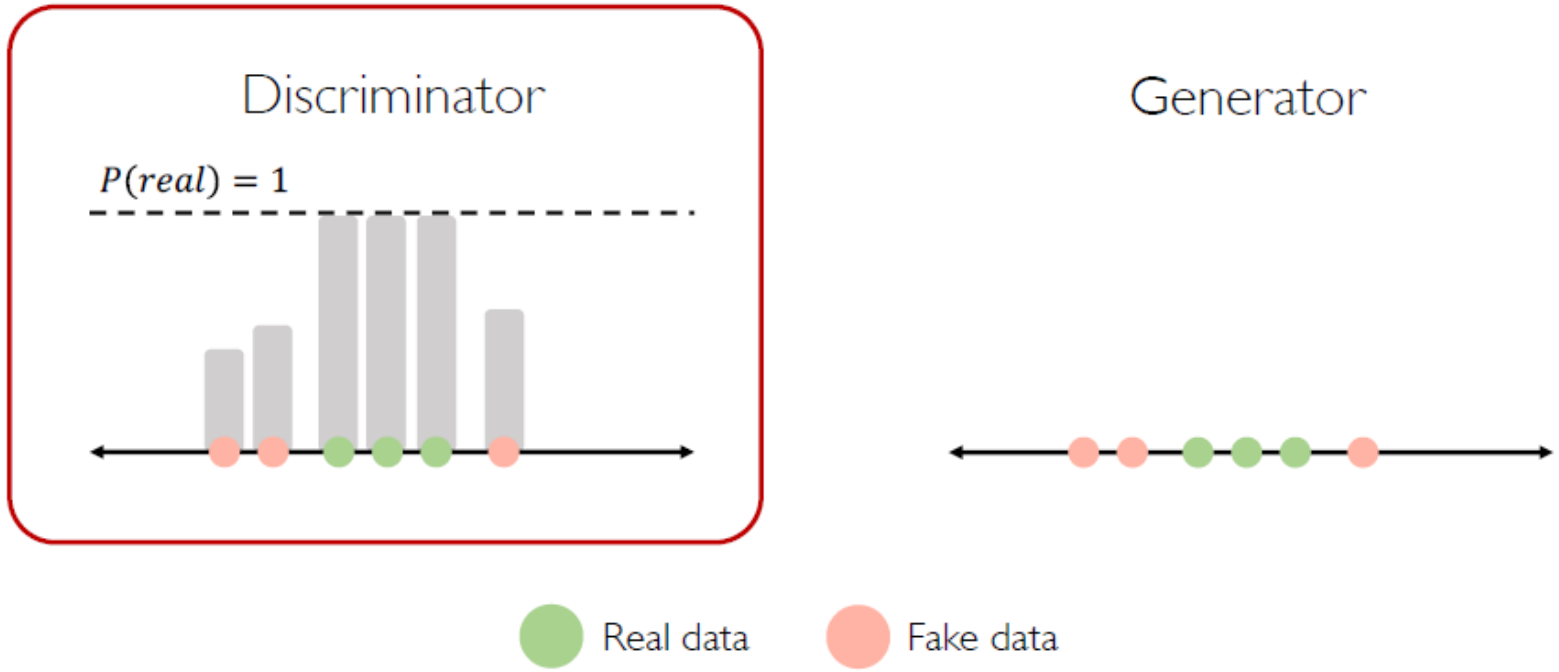
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

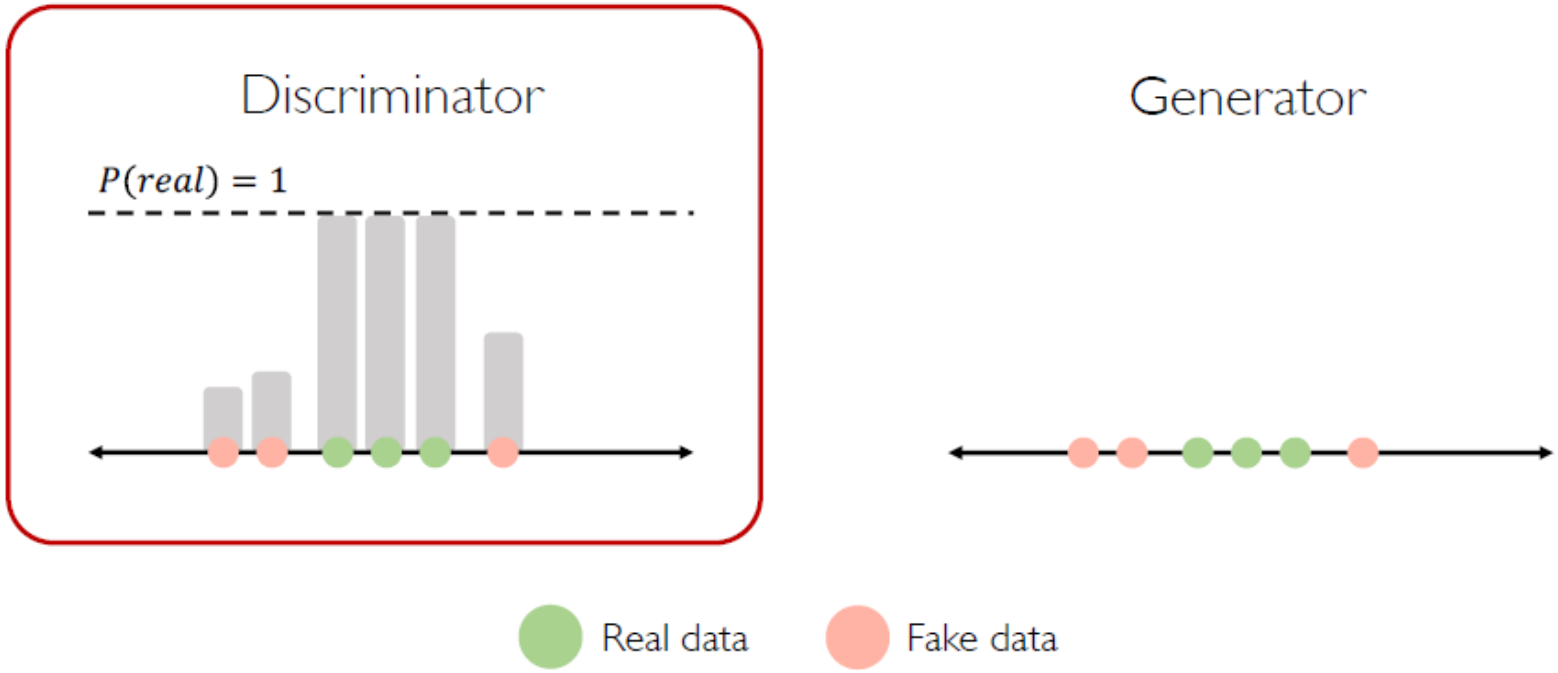
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

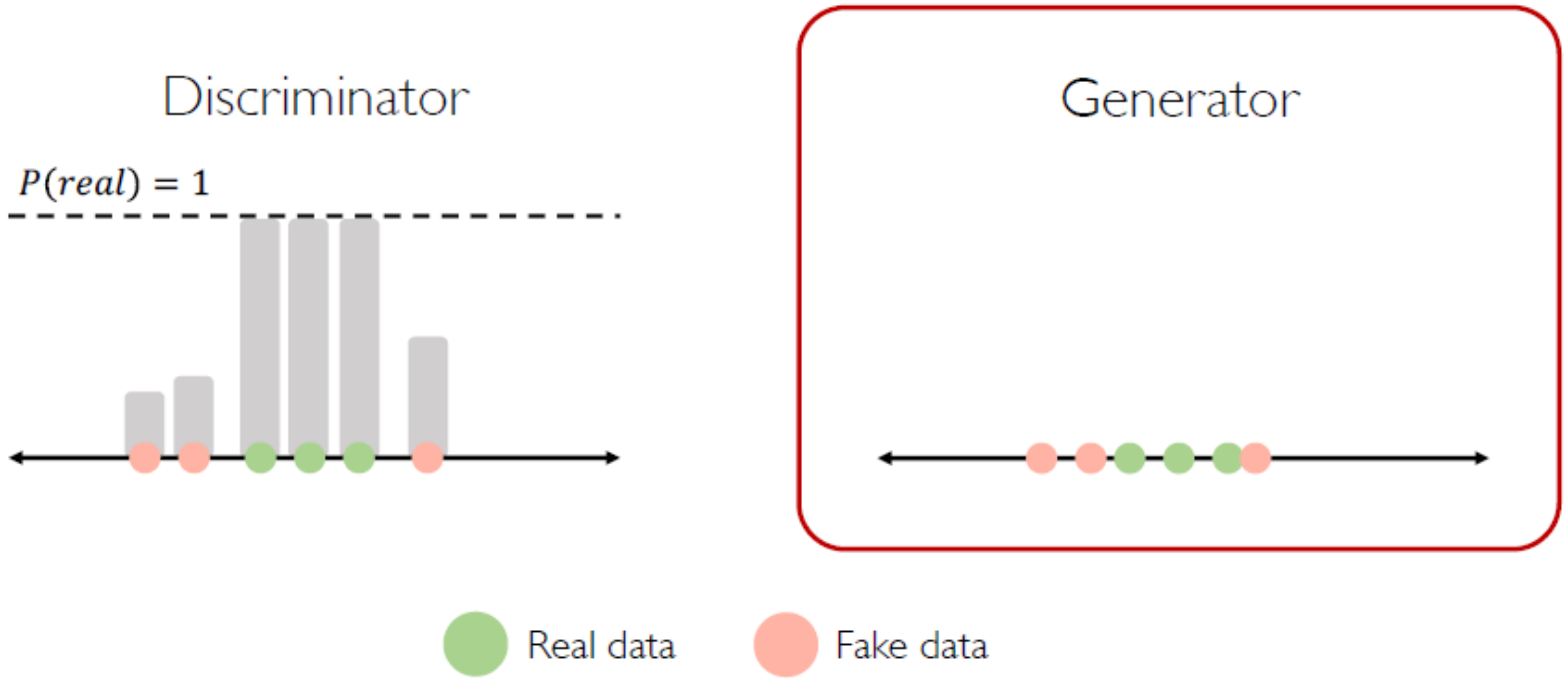
Discriminator tries to predict what's real and what's fake.



http://introtodeeplearning.com/

Intuition behind GANs

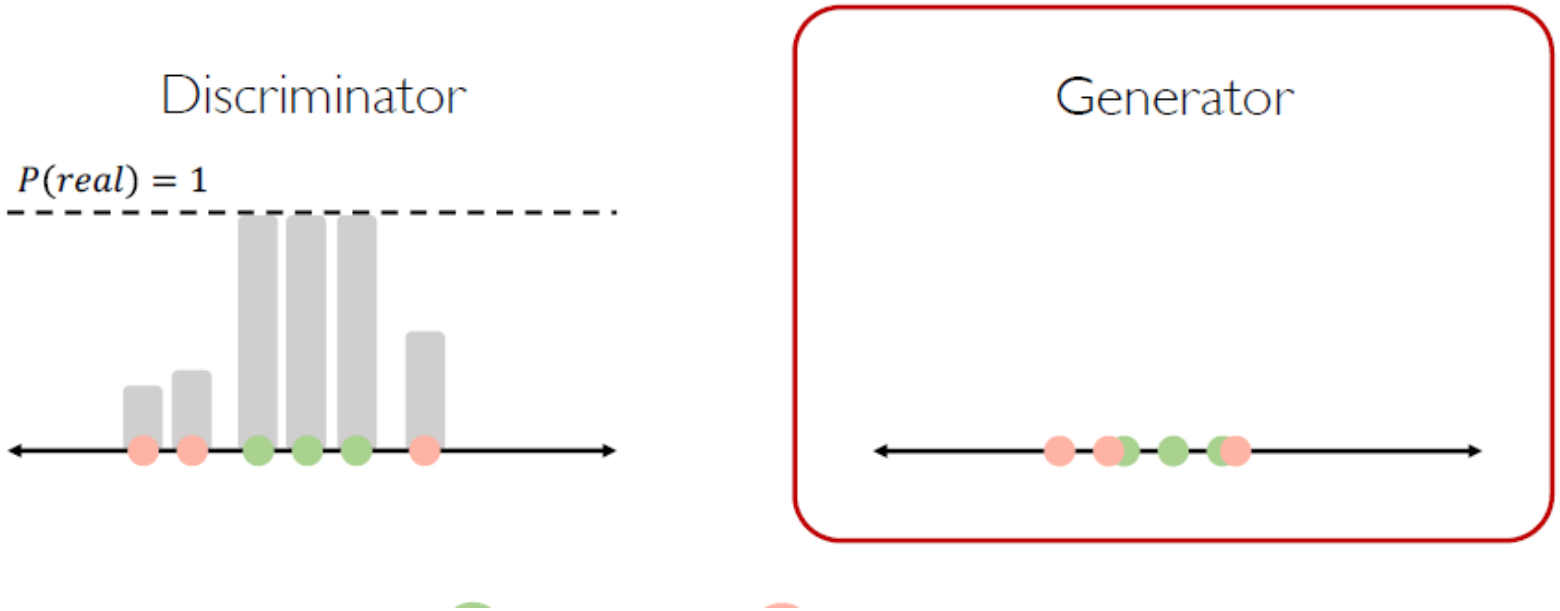
Generator tries to improve its imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

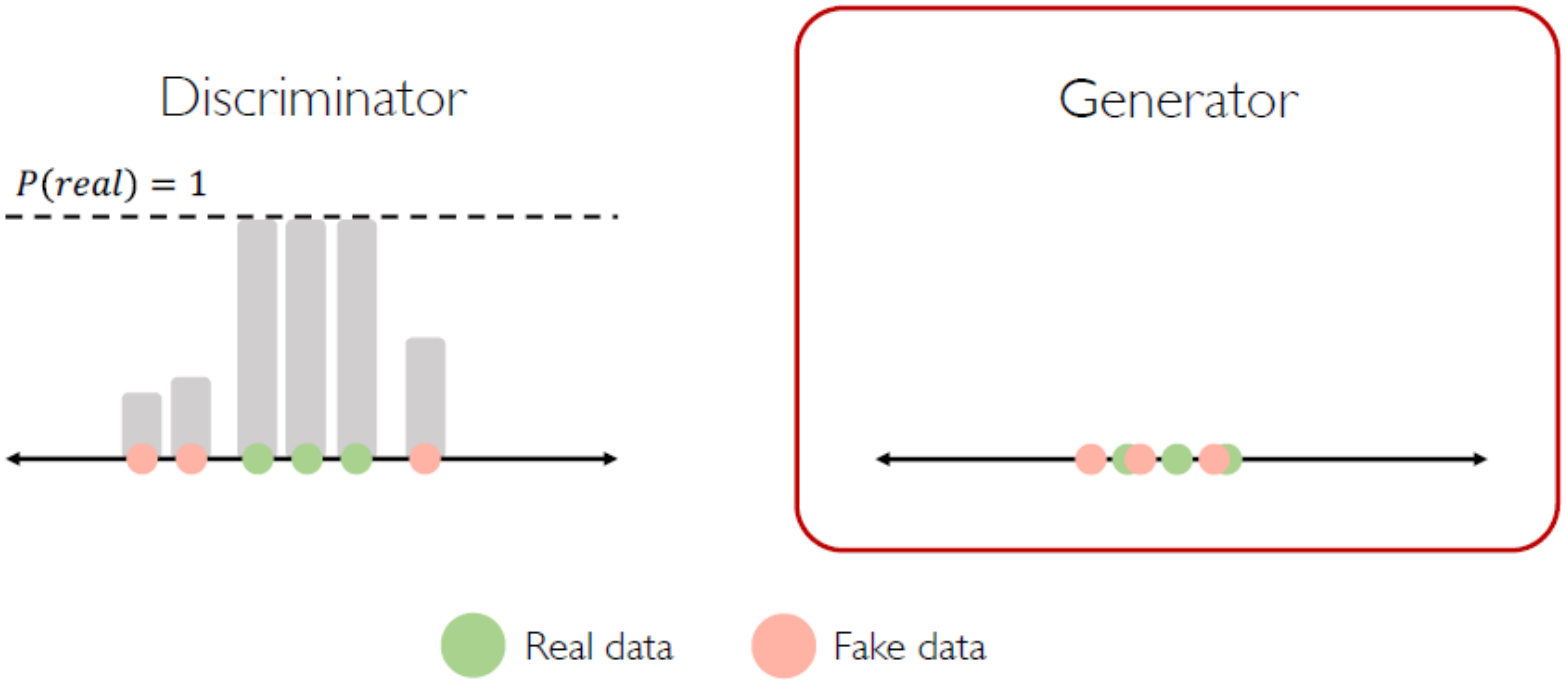
Generator tries to improve its imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

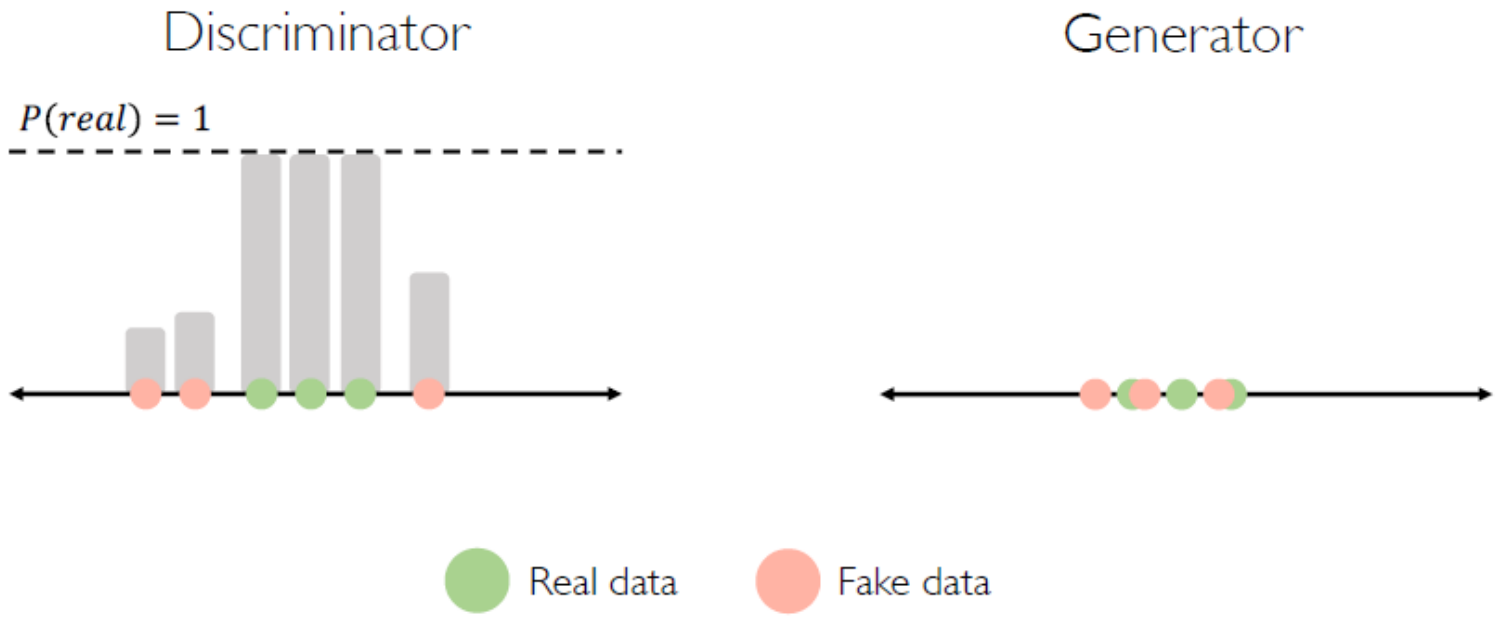
Generator tries to improve its imitation of the data.



http://introtodeeplearning.com/

Intuition behind GANs

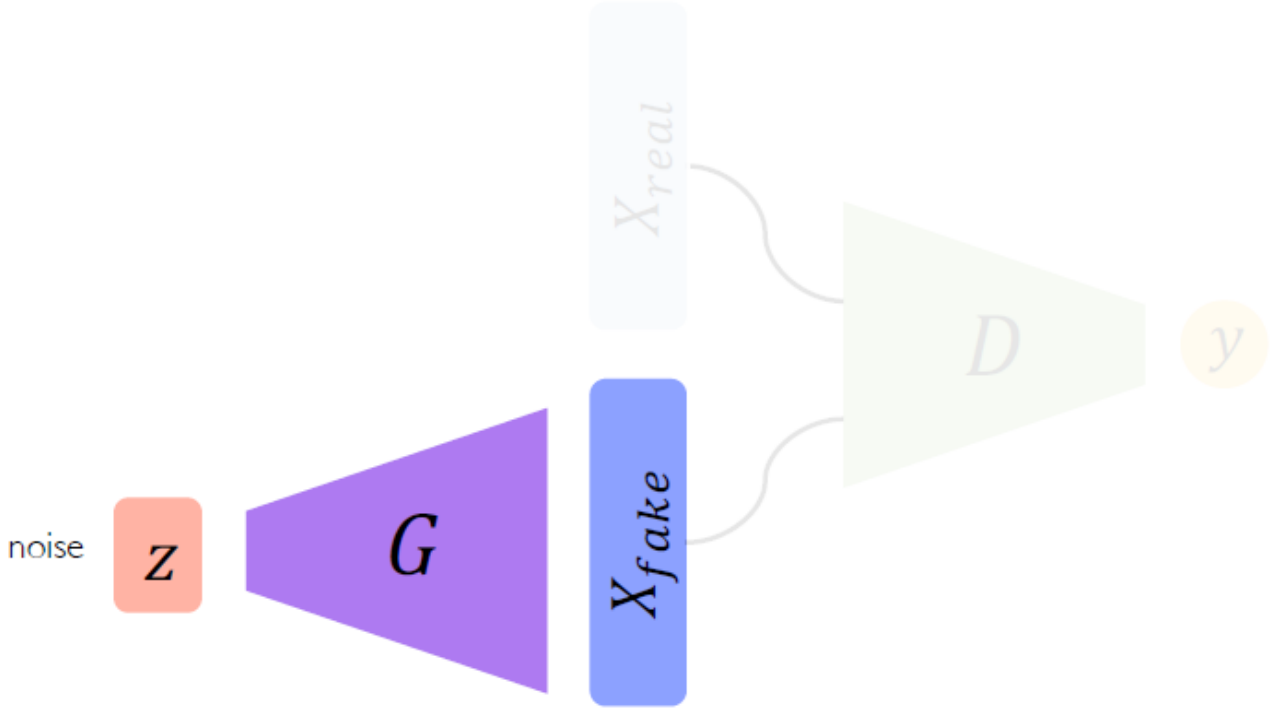
Discriminator tries to identify real data from fakes created by the generator.
Generator tries to create imitations of data to trick the discriminator.



http://introtodeeplearning.com/

Generating new data with GANs

After training, use generator network to create **new data** that's never been seen before.



http://introtodeeplearning.com/

Generative Adversarial Networks

- Makes neural networks compete against each other in the hope that this competition will push them to excel
  - A GAN is composed of two neural networks
- Capable of generating much more realistic and crisp images than variational autoencoders

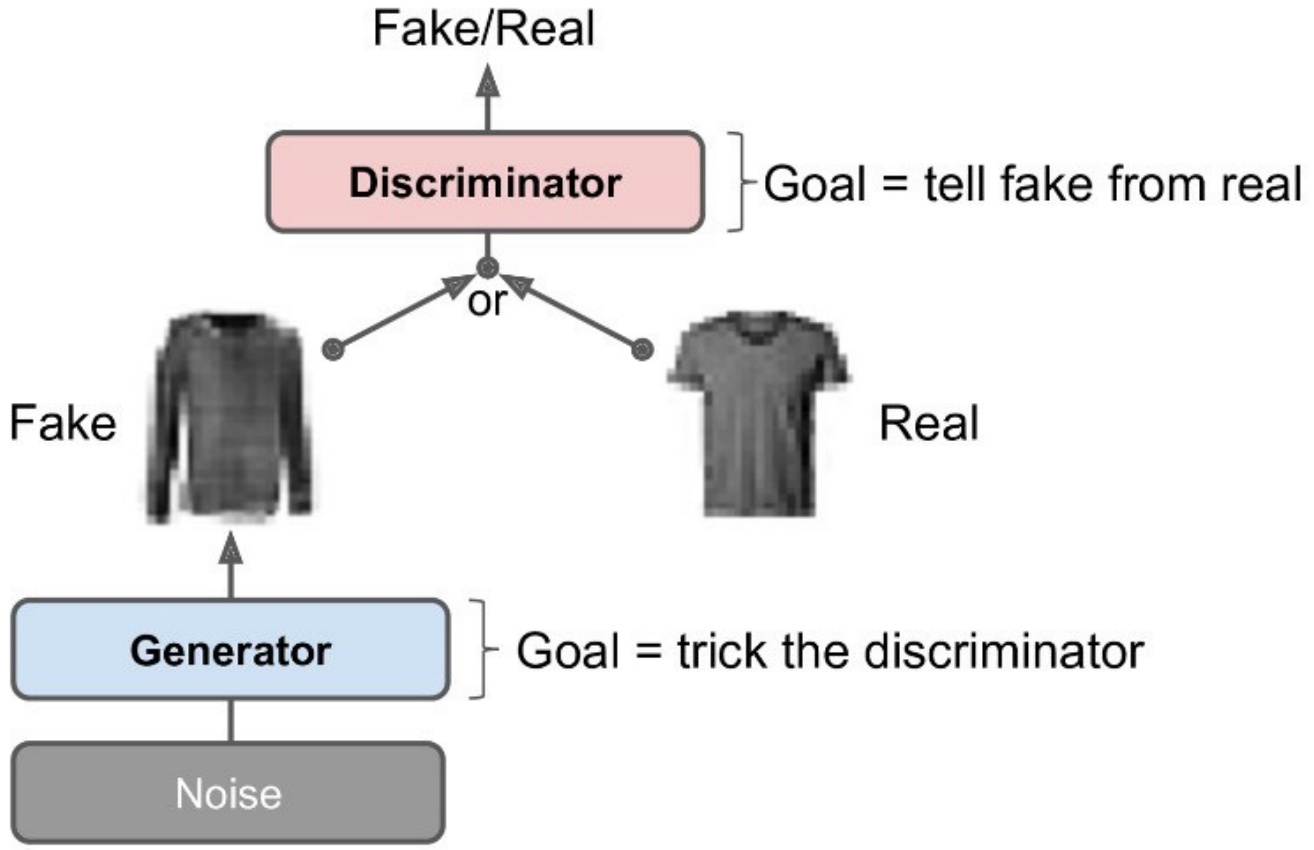


Figure 17-15. A generative adversarial network

A GAN is Composed of Two Neural Networks

Generator

- Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image
- You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated
  - offers the same functionality as a decoder in a variational autoencoder, and it outputs a brand-new image

Discriminator

- Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real

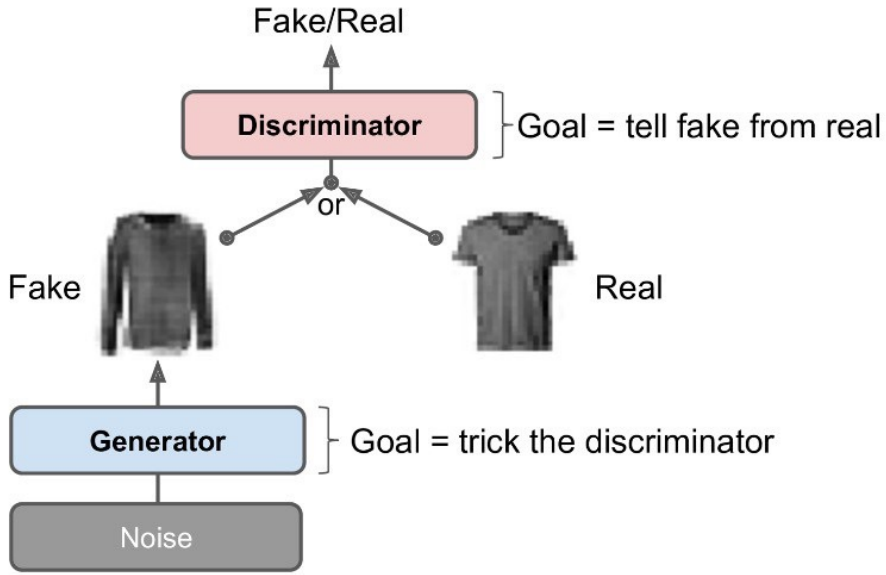


Figure 17-15. A generative adversarial network

Training a GAN

- The generator and the discriminator have opposite goals:
  - The discriminator tries to tell fake images from real images.
  - The generator tries to produce images that look real enough to trick the discriminator.
- The GAN cannot be trained like a regular neural network since it is composed of two networks with different objectives
- The generator never actually sees any real images, yet it gradually learns to produce convincing fake images
  - All it gets is the gradients flowing back through the discriminator
  - The better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress

Deep Convolutional GANS

- Alec Radford et al. deep convolutional GANs (DCGANs) in 2015 with the following guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use batch normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

Fashion MNIST Generated by DCGAN

Figure 17-16. Images generated by the DCGAN after 50 epochs of training

Vector Arithmetic for Visual Concepts

- Each of the three lower-left images represents the mean of the three images located above it
  - Not a simple mean computed at the pixel level (this would result in three overlapping faces)
  - It is a mean computed in the ***latent space***
  - ► *So the images still look like normal faces*
- If you compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the 3 × 3 grid of faces on the right: a woman with glasses!
- The eight other images around it were generated based on the same vector plus a bit of noise

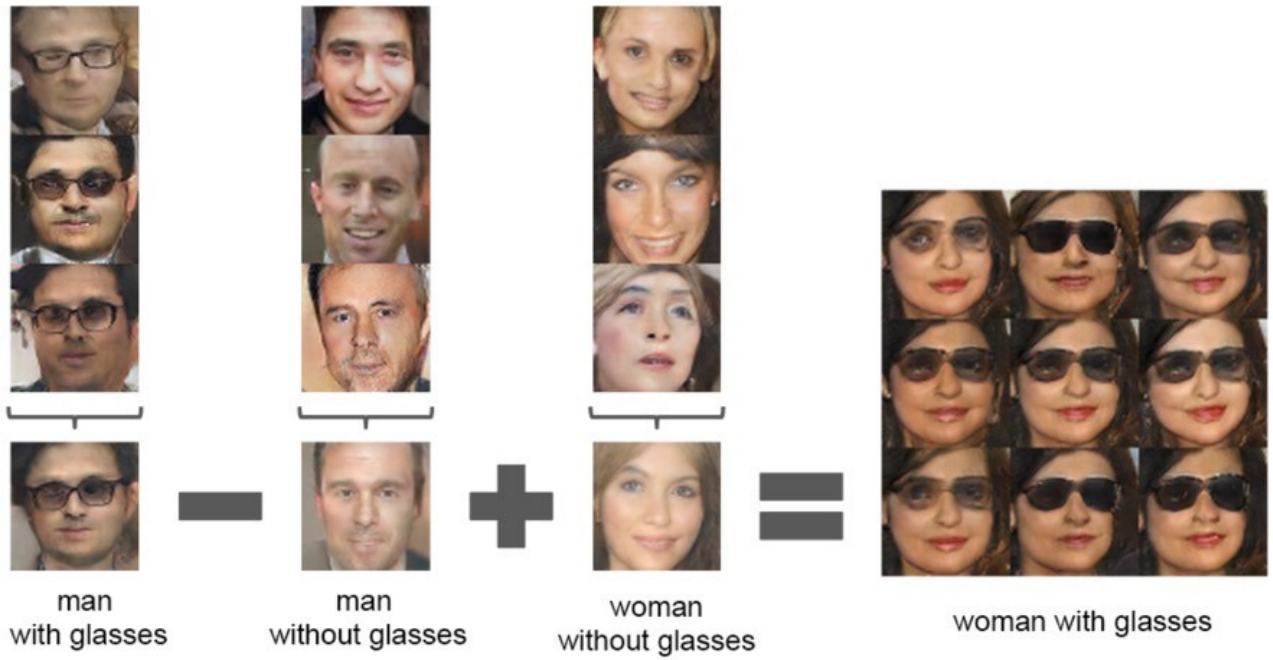


Figure 17-18. Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper)[14]

Outputs of a GAN

Karras et al., ICLR 2018.

http://introtodeeplearning.com/

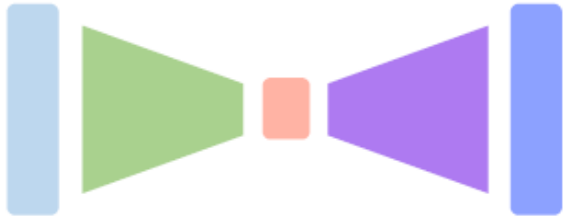
Style GAN

- The idea of adding noise independently from the codings is very important
- Some parts of an image are quite random, such as the exact position of each freckle or hair
- If this randomness came from the codings (and not independently from the codings):
  - Then the generator would have to dedicate a significant portion of the codings' representational power to store noise: this is quite wasteful
  - The noise would have to be able to flow through the network and reach the final layers of the generator: this seems like an unnecessary constraint that probably slowed down training.
  - Some visual artifacts may appear because the same noise was used at different levels.

Deep Generative Modeling: Summary

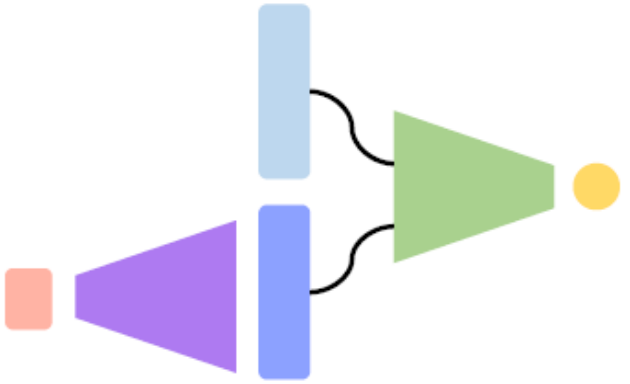
Autoencoders and Variational Autoencoders (VAEs)

Learn **lower-dimensional** latent space and **sample** to generate input reconstructions



Generative Adversarial Networks (GANs)

Competing **generator** and **discriminator** networks



http://introtodeeplearning.com/

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Homework Overview

Professor Hamza F. Alsarhan
SEAS 8505

This Week

- Reading:
  - Chapters 16 and 17 in the textbook
- HW #7 (Run/Write Python Script in Google Colab first, and then answer the homework questions)
- Discussion #7
- Start working on your final paper

- Reminder: No extensions provided. Start assignments early!

Next Steps

- Come to office hours with any questions you may have.
- Work on your HW and Discussion and submit them by 9:00 am ET on Saturday.
- See you next class!

Thank you!