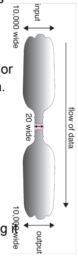
Ch 18 - Autoencoders



- The goal is to learn a compressed representation of data that captures its essential features while discarding noise or less relevant details.

Why would a biologist want to use an autoencoder?

1. **To reduce dimensionality for visualization purposes**.
A dataset might have thousands of features (or more). Reducing it to two (or a few) features makes it possible to capture general patterns found in data.
2. **To extract features.**
By learning a lower-dimensional representation of high-dimensional data, researchers can reduce noise and identify the most informative features, aiding in downstream analyses.
Such downstream analyses often include clustering and classification.
3. **To remove artifacts from images (and other data).**
When the data-collection process introduces variability or noise, autoencoders can learns to reconstruct the data without these, thus making it easier to identify main structures and patterns.
4. **To detect anomalies.**
Autoencoders can help to detect unusual patterns or rare events. During the data-reconstruction (decoding) process, an autoencoder might perform poorly for individual samples, thus helping researchers identify anomalies.

- Autoencoders are typically **lossy**, meaning that they do not perfectly reconstruct the original input data. Instead, they approximate it, often losing some details in the process, especially if the bottleneck layer has a low dimensionality.
- An autoencoder can be lossless when the bottleneck layer has a sufficient number of weights, essentially allowing the model to memorize and then perfectly reconstruct the training data.

But this approach is rarely used because it conflicts with the purpose of autoencoders.

What is the difference between **parametric** blending and **representation** blending?

Which do autoencoders use?

- In **parametric blending**, we learn information that describes the data abstractly (such as height, width, color, brightness).

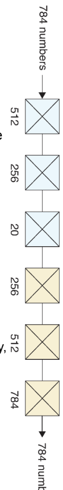
When creating new examples of data, we combine those parameters.

- In **representation blending**, we directly combine data from the objects we are blending.

Autoencoders typically use parametric blending. They learn a set of parameters (weights and biases). Through training, these parameters are optimized to minimize reconstruction error, allowing the model to learn meaningful data representations without explicitly blending representations from specific input samples.

Later we will learn about techniques that are more like **representation blending**:

- Variational autoencoders
- Generative adversarial networks



Latent Variables in an autoencoder are a compressed, abstract representation of the input data created in the bottleneck layer. Conceptually, latent variables capture the most essential features from the original data. "Latent" suggests that these patterns are inherent to the data, "just waiting for us to discover them."

Autoencoders are an example of **semi-supervised learning**. In what ways is this technique supervised? In what ways is it not?

- **Supervised**: The input data is essentially the target. The model optimizes for making the output be as similar as possible to the input.
- **Unsupervised**: There are no traditional targets, such as labels you would use in classification or continuous outputs you would use in regression.

Semi-supervised learning is often used in biology research when we have **lots of unlabeled samples and few labeled samples**.

The **supervised** part uses the labeled samples to learn relationship between features and target labels.

The **unsupervised** part looks for underlying structure in the data, helping to identify general patterns not specific to any labels.

Semi-supervised learning is different from transfer learning.

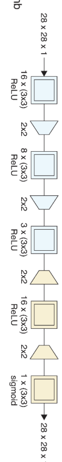
In semi-supervised learning, both labeled and unlabeled data relate to a specific research task.

Transfer learning uses data from a different domain or task to improve performance when you have limited labeled data for a specific research task.

The mean squared error (MSE) is typically used as the loss function for an autoencoder. Why do you think MSE is used?

- because it's simple, differentiable, and effectively measures reconstruction error by penalizing large deviations between input and output

Typical Architectures →



- Typically, when we use fully connected layers in an autoencoder, the number of neurons per layer decreases as we approach the bottleneck layer and then increases back to the size of the inputs. **(First architecture design above)**

- We may use a similar approach in a convolutional autoencoder. However, there is a wide variety of architectures. (**Second design to the right)**
- It is also possible to flatten images at the beginning of a neural network and then use fully connected rather than convolutional layers. But this approach does not take spatial information into account.

Denoising Autoencoders

When training a denoising autoencoder, you randomly add noise to the data / images. Then you train the model using the noisy data as the inputs and the original data as the outputs.

Conceptually, what will this teach the network to do?

This training process teaches the network to recognize and filter out noise from the input data, focusing instead on reconstructing the essential features and structure of the original, clean data.

Example for MNIST images

Step 1: Load and scale the images.

```
library(keras)
mnist = dataset_mnist()
x_train = mnist$train$x
x_test = mnist$test$x
# Rescale the data to values between 0 and 1
x_train = x_train / 255
x_test = x_test / 255
```

Step 2: Reshape the images to add a channel dimension.

```
x_train = array_reshape(x_train, c(nrow(x_train), 28, 28, 1))
x_test = array_reshape(x_test, c(nrow(x_test), 28, 28, 1))
```

We use **1 channel** for the dimension bc its **B&W**

Step 3: Add random noise.

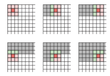
```
noise_factor = 0.5
x_train_noisy = x_train + noise_factor * array(rnorm(prod(dim(x_train))),
dim = dim(x_train))
x_test_noisy = x_test + noise_factor * array(rnorm(prod(dim(x_test))), dim
= dim(x_test))
# Clip values to keep them between 0 and 1
x_train_noisy = pmin(pmax(x_train_noisy, 0), 1)
x_test_noisy = pmin(pmax(x_test_noisy, 0), 1)
```

Step 4: Declare the inputs.

```
input_img = layer_input(shape = c(28, 28, 1))
```

Using **layer_input()** rather than **keras_sequential()** gives us more flexibility in how we define the model and access its parts.

Step 5: Define the encoder.



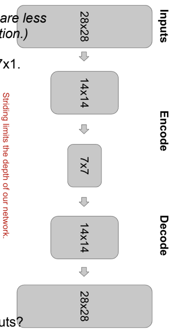
```
encoded = input_img %>%
  layer_conv_2d(filters = 32,
                kernel_size = c(3, 3),
                strides = c(2, 2),
                activation = "relu",
                padding = "same")
```

dimensions of the feature map after this step?

2x2 striding would reduce the dimensions from 28x28x1 to 14x14x1.

Step 6: Define the bottleneck layer.

```
bottleneck = encoded %>%
  layer_conv_2d(filters = 64,
                kernel_size = c(3, 3),
                strides = c(2, 2),
                activation = "relu",
                padding = "same")
```



(We use more filters than the previous layers because we are less concerned about compression and more about reconstruction.)

dimensions of the feature map after this step?

Striding would reduce the dimensions from 14x14x1 to 7x7x1.

Step 7A: Define the decoder.

```
decoded = bottleneck %>%
  layer_conv_2d_transpose(filters = 32,
                          kernel_size = c(3, 3),
                          strides = c(2, 2),
                          activation = "relu",
                          padding = "same") %>%
  layer_conv_2d_transpose(filters = 8,
                          kernel_size = c(3, 3),
                          strides = c(2, 2),
                          activation = "relu",
                          padding = "same") %>%
```

dimensions of the feature map after these steps?

Step 7B: Define the decoder.

```
layer_conv_2d_transpose(filters = 1,
                        kernel_size = c(3, 3),
                        activation = "sigmoid",
                        padding = "same")
```

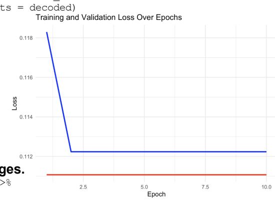
The number of **filters** is equal to the number of **channels**.

Effect the sigmoid activation function will have on the outputs?

The sigmoid activation layer ensures that the outputs fall between 0 and 1.

Step 8: Put the pieces together and compile.

```
autoencoder = keras_model(inputs = input_img,
                          outputs = decoded)
autoencoder %>%
  compile(optimizer = "adam",
          loss = "mse")
```



Step 9: Train the model.

```
history = autoencoder %>% fit(
  x = x_train_noisy,
  y = x_train,
  epochs = 10,
  batch_size = 128,
  shuffle = TRUE,
  validation_split = 0.2
)
```

Step 10: Denoise the test images.

```
denoised_images = autoencoder %>%
  predict(x_test_noisy)
```

What could we change to address the fact that the network stopped learning and produced images with all zeroes?

Some possibilities for regularization:

- Change the **activation function**.

```
activation = "leaky_relu"
```

(changed the activation function similarly for the other layers)

- Add **dropout**.

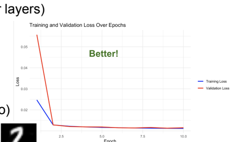
```
layer_dropout(rate = 0.25)
```

(added dropout to the bottleneck layer, too)

No better after these two changes

- Add **batch normalization**.

```
layer_batch_normalization()
```

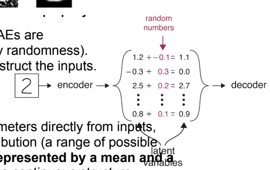


(added batch normalization to the bottleneck layer, too)

Better after this change:

Variational Autoencoders

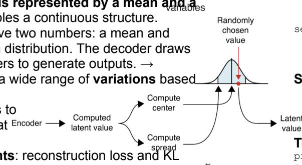
- Unlike simple autoencoders, VAEs are nondeterministic (are driven by randomness).
- They do not simply try to reconstruct the inputs.



- They generate something new every time.
- We **cannot** do this because backpropagation would not work properly.

Unlike simple autoencoders, which derive parameters directly from inputs, VAEs encode each input as a probabilistic distribution (a range of possible values) in the bottleneck layer. **Each input is represented by a mean and a standard deviation**, giving the latent variables a continuous structure.

- We generate latent variables that each have two numbers: a mean and standard deviation representing a Gaussian distribution. The decoder draws a random number based on those parameters to generate outputs. →
- Accordingly, VAEs enable us to **generate** a wide range of **variations** based on the inputs.



At the same time, the loss function attempts to constrain the model to generate outputs that are realistic according to the inputs.

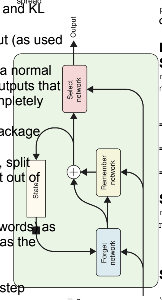
The **loss function** includes **two components**: reconstruction loss and KL divergence.

1. The **reconstruction** term ensures the output resembles the input (as used in simple autoencoders).
2. The **KL divergence** term encourages the latent space to follow a normal distribution. This regularization constrains the model to generate outputs that are consistent with—but different from—the inputs, rather than completely random outputs.

VAEs are much **more complicated** to implement with the Keras package than simple autoencoders.

- To avoid bias if your generating data for a classification analysis, split source dataset into training and testing, then generate new dataset out of those rather than splitting new dataset generated from VAE.

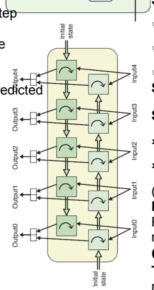
Ch 19 - Recurrent Neural Networks



This chapter focuses on using text (sequences of words) as inputs and predicting the next word for each input as the outputs.

Main ideas of RNNs:

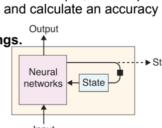
1. In a given layer of a network, we repeat the same step multiple times.
2. We keep a memory ("state") as we go through these repetitions so we can keep learning. (image below)
3. Eventually, we produce an output.



- After predicting the next word(s) for each input, we compare the predicted word against the actual next word and calculate an accuracy metric.

RNNs in Keras

Step 1: Prepare a vector of strings.




```
library(keras)
library(tidyverse)
my_strings = …
```

Step 2 - Clean the text.

- Convert to lower case.
- Remove extra white space.
- Remove numbers.
- Remove punctuation.

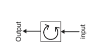
tm: Text Mining Package

Step 2.1: Install and load package

```
install.packages("tm") # Only need to run once
library(tm)
```

Step 2.2: Clean the text

```
abstracts = VCorpus(VectorSource(abstracts))
abstracts = tm_map(abstracts, content_transformer(tolower))
abstracts = tm_map(abstracts, stripWhitespace)
removeNumbers = content_transformer(function(x) gsub("\\b\\d+\\b", "", x))
abstracts = tm_map(abstracts, removeNumbers)
abstracts = tm_map(abstracts, removePunctuation)
```



gives us an object with a type that is specific to the tm package.

```
<<VCorpus>>
Metadata:  corpus specific: 0,document level (indexed):0
Content:  documents: 500)
```

Step 2.3: Convert back to a vector.

```
abstracts = sapply(abstracts, as.character)
# The sapply function acts like a "for" loop. It applies the specified
function to each element and then returns a vector.
```

Step 3.1: Create a tokenizer.

```
# This is the maximum number of (most frequent) words to use across all the
inputs.
max_words = 1000
tokenizer = text_tokenizer(num_words = max_words) %>%
  fit_text_tokenizer(abstracts)
```

Step 3.2: Create a tokenizer.

```
word_index = tokenizer$word_index
# This identifies all unique words and assigns an integer (token) to each.
print(head(word_index, n = 3))
Output:
$the  [1] 1  $and  [1] 2  $of  [1] 3
```

Inspecting the tokenizer.

```
print(head(tokenizer$word_counts, n = 3))
# This gives you a list that tells you how many times each word appears.
Output:
$health  [1] 214  $care  [1] 92 $has [1] 149
```

Step 4: Tokenize the text.

```
sequences = texts_to_sequences(tokenizer, abstracts)
print(head(sequences, n = 3))
```

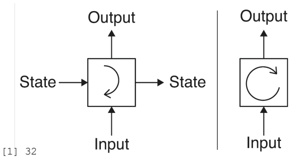
```
the aim of the current study was to
the aim of the current study was to
the aim of the current study was to
```

Step 5: Create input and output sequences.

```
input_sequences = list()
output_sequences = list()
seq_length = 5
for (sentence_seq in sequences) {
  if (length(sentence_seq) < seq_length + 1) {
    next}
  for (i in 1:(length(sentence_seq) - seq_length)) {
    seq_in = sentence_seq[i:(i + seq_length - 1)]
    seq_out = sentence_seq[i + seq_length]
    input_sequences[[length(input_sequences) + 1]] = seq_in
    output_sequences[[length(output_sequences) + 1]] = seq_out}}
```

Inspect input sequences:

```
head(input_sequences, n = 3)
Output:
[[1]]
[1]  60 169  99 366   5
[[2]]
[1] 169  99 366   5 218
[[3]]
[1]  99 366   5 218   9
```



Inspect output sequences:

```
head(output_sequences, n = 3)
Output:
[[1]] [1] 218 [[2]] [1] 9 [[3]] [1] 32
```

Step 6: Convert input sequences to a matrix.

```
input_sequence_matrix = pad_sequences(input_sequences,
  maxlen = seq_length,
  padding = 'pre')
head(input_sequence_matrix)
```

Step 7: Convert output sequences to a matrix.

```
# This performs one-hot encoding for each word
output_sequence_matrix = to_categorical(output_sequences,
  num_classes = max_words + 1)
print(output_sequence_matrix[1:6,1:6])
```

Now, this looks a lot like a multi-class classification problem.

Step 8.1: Build a simple model. **(image above)**

```
num_units = 128
model = keras_model_sequential() %>%
  layer_embedding(input_dim = max_words + 1, output_dim = num_units) %>%
```

The **layer_embedding()** function transforms integer-encoded words into continuous-valued vector representations. This allows each word to be represented by a lower-dimensional vector, based on patterns observed across the training set.

Step 8.2: Build a simple model.

```
num_units = 128
model = keras_model_sequential() %>%
  layer_embedding(input_dim = max_words + 1, output_dim = num_units) %>%
  layer_simple_rnn(units = num_units, dropout = 0.3) %>%
  layer_dense(units = max_words + 1, activation = 'softmax')
```

Given what you know about the outputs, which loss function should we use?

Step 9: Compile the model.

```
model %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = "adam",
  metrics = c('accuracy'))
```

Step 10: Fit the model.

```
history = model %>% fit(
  input_sequence_matrix,
  output_sequence_matrix,
  batch_size = 64,
  epochs = 10,
  validation_split = 0.2)
```

Step 11.1: Predict the next word for phrases.

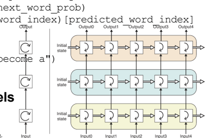
```
predict_next_word = function(seed_text) {
  # Encode and pad the input sequence
  encoded_sequence = texts_to_sequences(tokenizer, seed_text)[[1]]
  encoded_sequence = pad_sequences(list(encoded_sequence), maxlen =
seq_length, padding = 'pre')
    # Predict the next word probabilities
  next_word_prob = model %>%
    predict(encoded_sequence)
```

Step 11.2: Predict the next word for phrases.

```
  # Select the word with the highest probability (no randomness)
  predicted_word_index = which.max(next_word_prob)
  predicted_word = names(tokenizer$word_index)[predicted_word_index]
    return(predicted_word)}
```

Try it out with a phrase:

```
predict_next_word("health care has become a")
Output:
[1] "symptoms"
```



Building other types of RNN models

Step 8: Build a deep RNN model.

```
num_units = 128
model = keras_model_sequential() %>%
  layer_embedding(input_dim = max_words + 1, output_dim = num_units) %>%
  layer_simple_rnn(units = num_units, dropout = 0.3, return_sequences
= TRUE) %>%
  layer_simple_rnn(units = num_units, dropout = 0.3, return_sequences
= TRUE) %>%
  layer_simple_rnn(units = num_units, dropout = 0.3, return_sequences
= FALSE) %>%
  layer_dense(units = max_words + 1, activation = 'softmax')
```

Step 8: Build an LSTM model.

```
num_units = 128
model = keras_model_sequential() %>%
  layer_embedding(input_dim = max_words + 1, output_dim = num_units) %>%
  layer_lstm(units = num_units, dropout = 0.3) %>%
  layer_dense(units = max_words + 1, activation = 'softmax')
```

Step 8: Build a deep LSTM model.

```
  layer_lstm(units = num_units, dropout = 0.3, return_sequences = TRUE)
%>%
  layer_lstm(units = num_units, dropout = 0.3, return_sequences = TRUE)
%>%
  layer_lstm(units = num_units, dropout = 0.3, return_sequences = FALSE)
%>%
```

Step 8: Build a deep, bidirectional RNN model.

```
  bidirectional(layer_simple_rnn(units = num_units, dropout = 0.3)) %>%
```

Step 8: Build a deep, bidirectional LSTM model.

```
  bidirectional(layer_lstm(units = num_units, dropout = 0.3,
return_sequences = TRUE)) %>%
  bidirectional(layer_lstm(units = num_units, dropout = 0.3,
return_sequences = TRUE)) %>%
  bidirectional(layer_lstm(units = num_units, dropout = 0.3)) %>%
```

(Combine first and 3rd flow charts)

Building a Seq2Seq model is more complicated.

Rather than doing that, we will wait until the next chapter to use some models that are even more powerful.

Ch 20 - Attention and Transformers

Tokens

relationship between words and tokens?

Token: The basic unit of text that a language model processes. They can be **words**, **subwords**, or even **characters**, depending on the tokenizer used.

Subword tokenization: Break (some) words into smaller parts.

```
"unbelievability" →
 c("un", "##bel", "##ie", "##va", "##bility")
```

(The ## helps the model know which subwords belong together to form a complete word.)

- This can capture semantic nuances, such as prefixes, roots, or suffixes that contribute to the overall meaning. These often overlap across words.

When working with text that a model has not previously seen, subword tokenization can reduce the overall vocabulary size and the number of "unknown" tokens.

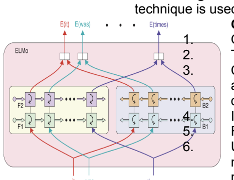
This is especially helpful for rare words.
Subword tokenization also helps with processing of (human) languages that have complex morphologies.
For example, in Finnish,
epäjärjestelmällistyttämättömyydellänsä
is a highly inflected word that means something along the lines of "with their lack of organization."
Subword tokenization can help to break such words down and account for the ways inflection is used.
One token = one character.
Advantage: It can handle rare words, typos, and many languages consistently.
Disadvantage: It requires more computational resources and may not be able to decipher complex semantics.
Modern language models rely mostly on **subword tokenization.**
relationship between tokens and embeddings?
- Tokens are the basic units of text used in a model, whereas **embeddings are numerical-vector representations of those tokens**, enabling the model to understand and manipulate text computationally.
Embeddings can be derived from individual words (after tokenization) or from sentences.
Sentence-level embeddings are common, but **we will focus mostly on word-level embeddings** for simplicity.
By training embeddings on particular inputs, the goal is that words/tokens with similar semantic meanings or that play similar contextual roles have embeddings that are close (mathematically) in the embedding space.
For example, we would expect the embeddings these word pairs to close in the embedding space:
- Disease ↔ Illness
- Doctor ↔ Physician

But less so for these:
- Neuron ↔ Chloroplast
- Hemoglobin ↔ Leaf

With high-quality embeddings, we can make interesting inferences.
`King - Man + Woman = Queen`
By average embeddings for two words, we might obtain an embedding that is conceptually "between" those words.
For example, if we averaged embeddings for **winter** and **summer**, we might get **spring**.
High-quality embeddings are essential to performing a range of tasks. Such as:
- **Sentiment analysis**. Evaluating whether the tone of a sentence is positive or negative.

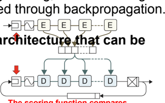
This could be useful to biologists for tasks like:
- Analyzing public perceptions of science topics.
- Evaluating patient feedback in medical studies.
- Monitoring emotional well-being in the behavioral sciences.
- Classifying human annotations of animal behavior.
- **Named entity recognition.** Assigning categories to words or phrases in a sentence.

This could be useful to biologists for tasks like:
- Mining journal articles for gene-disease or protein-disease associations.
- Extracting lab results, symptoms, or treatment names from clinical trial reports.
- Detecting mentions of species, habitats, or ecological terms in biodiversity reports.
- Extracting chemical compound names and their interactions with proteins or diseases from text.
- **Summarizing text**. Distilling a large body of text into a coherent summary.
- **Question answering.**
- **Translating text from one (human) language to another.**
- **Semantic search.** Finding similar documents or sentences. May be useful for search engines and/or clustering.
- **Generating text**. Creating new text based on prompts. This technique is used by chatbots.

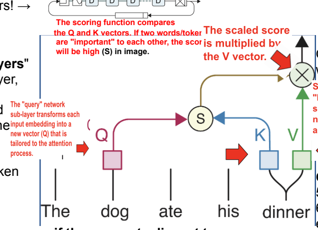


Creating Embeddings
1. Obtain a training "corpus".
2. Tokenize the text.
3. Create a matrix where each row represents a token and each column is an embedding dimension.
4. Initialize the values to random numbers.
5. Feed the matrix into a neural network.
6. Use a task-specific loss function to train the network. During training, the embedding matrix is updated through backpropagation.

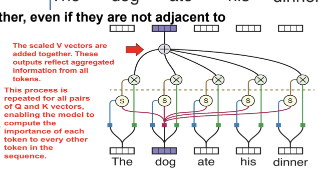
Later we'll learn more about such loss functions →
(above) ELMo is a recurrent neural network architecture that can be used to train embeddings.

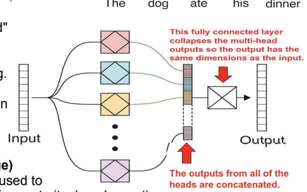


But ELMo has largely been replaced by attention-based models like transformers! →
Attention
A softmax layer scales the scores.
This keeps the scores from getting too large or small
(Line between S and X)
- We are calling these "**network sub-layers**" because they are part of an attention layer, which is part of a transformer network.
- They have weights, which are adjusted according to a loss function that is defined elsewhere in the overall network.
Importance can be defined in different ways, but typically it means that one token provides relevant information for understanding the meaning or role of another token within the sentence.

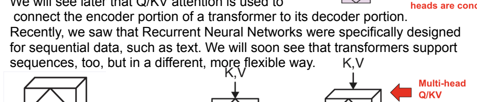


Tokens may be important to each other, even if they are not adjacent to each other in a sentence!
What does multi-head attention (bottom left) do that basic self attention does not?
Multi-head attention splits the attention mechanism into multiple, independent "heads," each with its own set of learnable parameters.
Example: "He sat by the bank and watched the duck swim."
Head 1: Focuses on "sat" and "watched" to understand the activity context.
Head 2: Focuses on "duck" and "swim" to reinforce the idea of a natural setting.
Head 3: Captures sentence structure, identifying that "bank" is likely a location rather than a financial institution.
Q/KV attention uses one source for the queries and another for the keys and values. (Bottom right image)





We will see later that Q/KV attention is used to connect the encoder portion of a transformer to its decoder portion.
Recently, we saw that Recurrent Neural Networks were specifically designed for sequential data, such as text. We will soon see that transformers support sequences, too, but in a different, more flexible way.



Hugging Face
The company was founded in 2016…originally as a company that developed a chatbot app targeted at teenagers. The company was named after the… 🤗 HUGGING FACE emoji. After open sourcing the model behind the chatbot, the company pivoted to focus on being a platform for machine learning.
Let's cover how to do a few types of tasks with Hugging Face:

- Classifying text (sentiment analysis).

```
library(reticulate)
transformers = reticulate::import("transformers")
sentiment_analyzer = transformers$pipeline(task = "text-classification")
print(sentiment_analyzer("I love science"))
```

- Summarizing text.

```
summarizer = transformers$pipeline("summarization")
print(summarizer("text…", max_length = 56L))
```

- Question answering.

```
answerer = transformers$pipeline(task = "question-answering")
question =  "Which disease is this about?"
text = "We present the molecular landscape of pediatric acute myeloid
leukemia (AML) and characterize nearly 1,000 participants..."
print(answerer(question = question,
               context = text,
               clean_up_tokenization_spaces = TRUE))
```

- Translating text. (english → french)

```
translator = transformers$pipeline(
  task = "translation_en_to_fr",
  model = "Helsinki-NLP/opus-mt-en-fr")
print(translator(
  "How old are you?",
  clean_up_tokenization_spaces = TRUE))
```

3.29	4.68	1.49	0.02

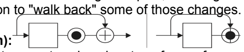
+

-1.45	2.29	4.67	1.92

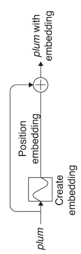
=

1.84	6.97	6.16	1.94

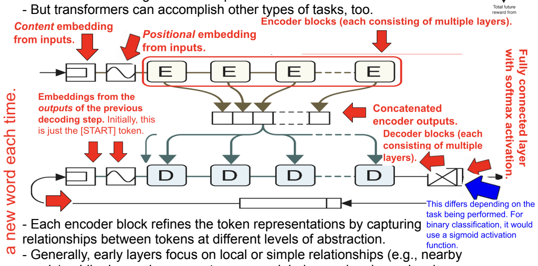
← Skip (residual) connections: →
1. The model learns something from the data, spanning multiple layers.
2. The skip portion adds the inputs to the outputs.
if recent updates to a neural network' layers do not effectively minimize the loss, skip connections preserve access to the original inputs, ensuring the network retains critical information to "walk back" some of those changes. (hiker trail marker analogy)



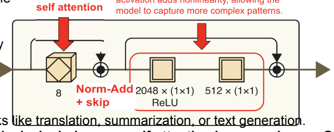
Norm-Add (layer normalization):
- This is a regularization step that prevents values in a transformer from getting too large or too small.
- It adjusts the values to have a Gaussian distribution with a mean of zero and standard deviation of one.
- Often we place it right before a skip connection.



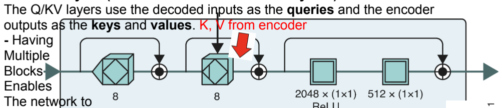
Positional Encoding
What? A way for transformers to account for positional information in a sequence of inputs.
Why? Sequence matters, but we also need flexibility to account for long-range dependencies between tokens and phrases that are structured differently but have the same meaning.
← How? Encode the position information in an embedding that is the same size as the input embedding. This embedding is added to the input. The combined embedding reflects both sources of information.
Transformer Architecture
- Frequently, a transformer's goal is to predict the next word in a sequence.
- We'll focus on that goal in this explanation.
- But transformers can accomplish other types of tasks, too.



- Each encoder block refines the token representations by capturing relationships between tokens at different levels of abstraction.
- Generally, early layers focus on local or simple relationships (e.g., nearby words), while deeper layers capture more global, complex dependencies.
Zooming in on an encoder block:
- Stacking multiple decoder blocks increases the model's capacity to learn and represent the intricate Relationships required for tasks like translation, summarization, or text generation.

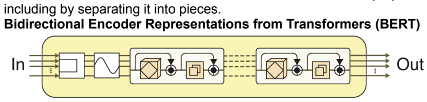


- The decoder blocks include some self-attention layers and some Q/KV attention layers ("encoder-decoder attention layers").
The Q/KV layers use the decoded inputs as the **queries** and the encoder outputs as the **keys** and **values**. K, V from encoder
- Having Multiple Blocks Enables The network to Find more nuanced mappings between the input seq and the gen output.

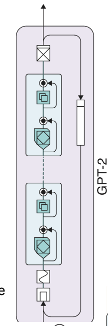


- Because the decoder uses Q/KV attention, it can "pay attention" to the encoder outputs in any order.
- How might this be helpful?

One task where this is helpful is **translating from one language to another,** where phrases may be structured quite differently for the two languages.
Other tasks where this is helpful are document summarization and question answering. Being able to connect concepts expressed in different parts of the text enables more effective understanding and generation of contextually relevant outputs.
Training the whole network
Creating embeddings:
5. Feed the matrix into a neural network.
6. Use a task-specific loss function to train the network. During training, the embedding matrix is updated through backpropagation. Examples include:
- Next word prediction (self-supervised learning)
- During training, the first attention layer in each decoder block can be executed in parallel.
- Each parallelized task is given the same target sequence and is asked to predict a particular token in the target. Tokens after the target are masked.
Ex) `Plants grow best [mask] [mask] [mask] [mask].`
`Plants grow best when [mask] [mask] [mask].`
`Plants grow best when given [mask] [mask].`
- These tasks are executed in parallel, in no particular order.
- Masked token prediction (self-supervised learning)
Ex) `the aim of the current study was to compare physical [masked] and sleep duration between patients with chronic fatigue`
- This is different from masked multi-head self attention because we're not trying to predict the next word. We're just filling in missing words.
- Contrastive learning
Goal: Move semantically similar sentences closer in the "embedding space" and push dissimilar ones apart.
Similar: How do I bake a chocolate cake?
If I wanted to bake a chocolate cake, how would I do it?
Dissimilar: What are the benefits of exercising regularly?
What are the benefits of keeping a pet dragon?
Transformers were originally designed for language translation. But soon researchers realized that transformers could be used for other purposes, including by separating it into pieces.
Bidirectional Encoder Representations from Transformers (BERT)



ELMo - stacked bidirectional LSTMs, process the input in both directions. However, no direct interaction among tokens, other than for adjacent tokens.
BERT - Transformer architecture (encoder only), which incorporates multi-head self-attention. This allows every token in the input sequence to attend to every other token in both directions.
Also differences in training-corpus sizes, loss functions, and # of layers that differ
Generative Pre-Training (GPT)
How GPT models differ from BERT:

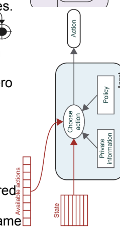


- **Architecture**: It only uses the decoder portion of the transformer architecture.
- **Unidirectional**: It processes tokens sequentially from left to right.
- **Training objective**: Predict the next token in a sequence based on previous tokens (self-supervised learning).
- Commonly fine tuned for specific objectives.

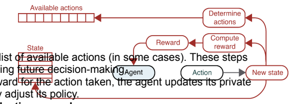
Because GPT **only** has an decoder, it is primarily designed for generation tasks, such as conversational AI (chatbots), text summarization, image generation, code generation, and new content generation.
Zero-shot learning: The model performs a task without any specific examples or prior task-specific fine-tuning, relying solely on its pretrained knowledge.
One-shot learning: The model performs a task after being given only a single example to understand the task.
Few-shot learning: The model performs a task after being provided with a small number of examples or prompts to understand the task context.

Ch 21 - Reinforcement Learning

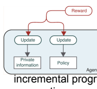
← The agent attempts to select the most appropriate action for a particular situation. It chooses the action by considering the "State" (environment context), the "Available actions" (possible options), "Private information" (internal data or preferences), and the "Policy" (decision-making rules or strategy).



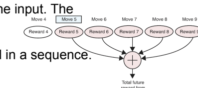
How the environment responds to an agent's action →
After the agent selects an action, the environment stores the new state information, determines what the reward should be, and modifies the list of available actions (in some cases). These steps provide feedback to the agent for improving future decision-making.



- After receiving a reward for the action taken, the agent updates its private information and may adjust its policy.



Allocating and evaluating rewards
- Depending on how the reward process is designed, positive rewards can be given for actions or outcomes that represent progress toward the goal, even if the task itself is not yet complete.
- In cases where task completion is considered to be more important than incremental progress, a positive reward is only given when the end goal has been reached. Zero or negative rewards might be given for steps taken before the end goal is reached.
- If the "Reward" is poorly defined or misinterpreted, the agent may **optimize for unintended behaviors, fail to achieve the desired goal, or exploit loopholes** in the reward system.This can lead to inefficiency, unsafe actions, or behavior that prioritizes short-term rewards over long-term objectives.
- When deciding which action to take, the agent must estimate the **total future reward** that will result from each *available action*. It uses its current policy and private information to estimate these. **(up left figure)**



Example: A predator-prey system
State: Location and behavior of the predator and prey; environmental conditions.
Available actions: Stalk, chase, ambush (hunting strategies).
Private information: The predator's current energy level and prior experiences.
Policy: The predator's "strategy" for choosing the most effective hunting method at any given time.
Q-Learning
Q-Learning is based on the premises that 1) **there is uncertainty in most systems** and 2) **estimating future rewards is challenging**.
- It focuses on the current state, the immediate reward, and the expected future rewards, using an iterative process.
- This process refines estimates of long-term rewards by creating a Q-table and updating it over multiple episodes.

We take the selected action, and the environment returns a reward and a new state.
The next step is to update the Q-value for the original state.
Suppose this is what we know so far:

```
original_state_q = 0.9
new_state_q = 0.5
reward = 0.54
gamma = 0.8
```

We also need to specify alpha, α, the learning rate.
This hyperparameter falls between 0 and 1.
For relatively high α values, more weight is given to new information.
For relatively low α values, less weight is given to new information. - Let assign `α = 0.9`.

A **high α** may be appropriate in the early stages of learning and in fast-changing environments.
A **low α** may be appropriate in stable environments or later stages of training, where the agent has already learned reasonable estimates and needs to fine-tune its Q-values.
We calculate a new Q-value for the original state.
Next, we update the Q-value.
This process repeats many times. The Q-table is iteratively updated as the agent selects actions, observes the environment's responses, and receives rewards. It (hopefully) improves its ability to choose optimal actions.
But one part is missing!
Epsilon-greedy policy
The **explore or exploit dilemma** states that there is a conflict between:
Explore: Trying something new that might lead to failure or might be even more successful
Exploit: Playing it safe by choosing actions that we have already seen to be successful.
In Q-learning, the epsilon-greedy policy allows us to specify how much we want to favor exploration or exploitation.
- **Higher values of epsilon, ϵ, favor more exploration.**
- **Lower values of epsilon, ϵ, favor more exploitation.**

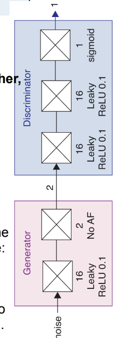
But let's say ϵ is 0.01. This means we want to explore a little.

```
if (runif(1) < 0.01) {
  # Randomly select an action.
} else {
  # Select action with the highest Q-value.}
```

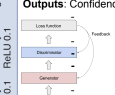
Deep Q-Networks (DQNs)
So far **none** of these steps have used machine/deep learning.
Storing a Q-table in memory is infeasible for tasks with a large number of states and/or actions.
Deep Q-Networks do not store a Q-table explicitly.
Instead, they use a neural network to generate an approximate Q-value function. The network takes the current state as input and outputs Q-values for all possible actions. The 'logic' for estimating Q-values and choosing actions is encoded in the network's weights, which are learned through experience but do not reflect every possible combination of state and action that could occur.
In DQNs, α is typically not explicitly included in the Q-value update logic, because it is inherently part of the backpropagation process used to train the neural network. Instead, you can change the optimizer's learning rate.

```
optimizer_adam(learning_rate = 0.001)
```

Ch 22 Generative Adversarial Networks (GANs)



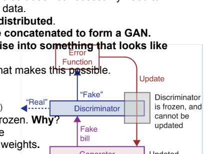
Generator - Inputs: Noise (random numbers) - **Outputs:** Sample
Discriminator - Inputs: Samples (originates either from generator (fake) or training set(real)) - **Outputs**: Confidence that the sample is real



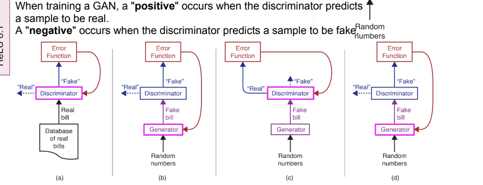
- The distribution (shape, mean) of the noise does not necessarily need to (and often does not) resemble the real data.
- Often the noise values are **uniformly distributed**.
- **The generator and discriminator are concatenated to form a GAN.**
- **The GAN learns to transform the noise into something that looks like the "real" training data.**

← A loss function provides feedback that makes this possible.

```
discriminator %>% compile(
  optimizer = "adam",
  loss = "binary_crossentropy")
```



When the generator is being updated, the discriminator is frozen. **Why?**
This ensures that the discriminator doesn't interfere with the backpropagation process when optimizing the generator's weights.
Training a GAN
When training a GAN, a "**positive**" occurs when the discriminator predicts a sample to be real.
A "**negative**" occurs when the discriminator predicts a sample to be fake.



This process continues for many rounds (epochs) of learning.
After the full GAN has been trained, the generator is used on its own. (X the discriminator from the figure up right)
Autoencoders vs. GANS
Similarities:
- Both can learn complex patterns from input data.
- Both can generate new samples that resemble the training data.
- Both are often used in unsupervised learning tasks.
Differences:
Training approach:
- Autoencoders are trained to minimize the difference between the input and output, using reconstruction loss.
- GANs are trained adversarially, starting from random noise, using a generator and discriminator.
Applications:
- Autoencoders encode data into a compressed latent space, which is often used for clustering, anomaly detection, dimensionality reduction.
- GANs are used to generate entire new variations of the training data.
Applications of GANs
Biology application: Creating high-resolution microscopy images from low-resolution inputs (super-resolution imaging).
Medical application: Generating realistic but anonymized medical data for training models without exposing sensitive patient information.