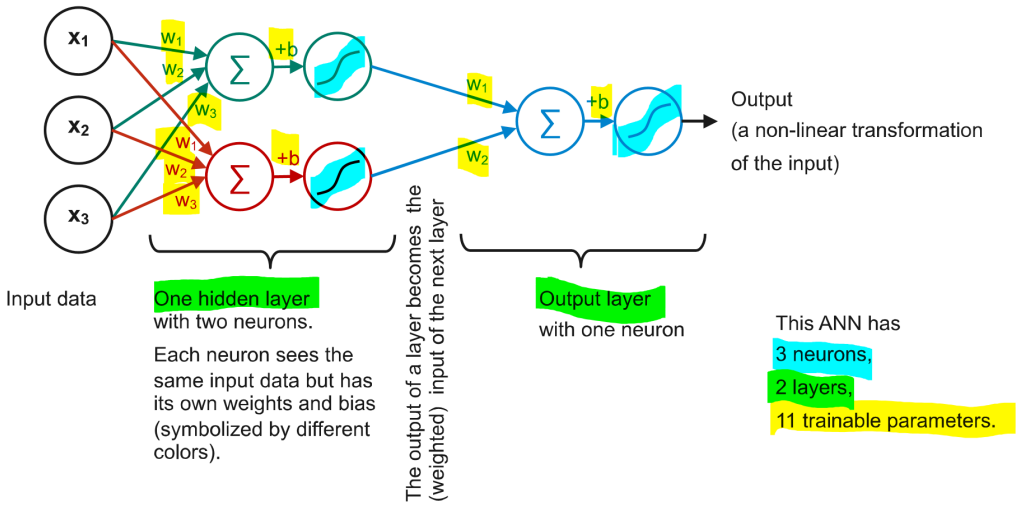
AI Applications | AIAp

Zusammenfassung

CONTENTS

1. ARTIFICIAL NEURAL NETWORK (ANN)

An ANN is a machine learning model inspired by the structure and function of animal brains. It consists of ***neurons*** inside a ***layer***. Neurons receive, process and send signals to other neurons in the next layer. The output is computed by some non-linear function of the sum of its inputs, the ***activation function***. The strength of the signal at each connection is determined by a ***weight***, which adjusts during the learning process. Signals travel from the ***input layer*** to the ***output layer***, while possibly passing through multiple intermediate layers, the ***hidden layers***.



3-dimensional input x_n, hidden layer with ***2*** neurons, output layer with ***1*** neuron. The sum of all inputs with its weights W_n for each input is added to a bias b before it is passed to the corresponding neuron.
Trainable parameters: $2 \cdot (3 \text{ weights} + 1 \text{ bias}) + 1 \cdot (2 \text{ weights} + 1 \text{ bias}) = 11 \text{ parameters}$

2. LARGE LANGUAGE MODEL (LLM)

A LLM is a ***huge ANN*** with a ***particular structure*** that is much more complicated. A LLM generates text ***sequentially*** *(word-by-word or token-by-token)*. The output is not a word, but a ***probability distribution over a dictionary***. The next word is sampled according to this distribution and is the most "plausible" according to the model. ***auto-regressive:*** each output is added to the input-context. So the generation of each next word takes into account the entire sequence generated so far.
The ***knowledge of an LLM*** is represented in the ***weights*** and the ***network structure***. Data is not stored explicitly.

2.1. PROMPT ENGINEERING

A LLM performs better when it has access to ***more context***. A general, open ended prompt will produce a worse output than one that provides information about the ***starting point*** and the ***desired result***, often divided in the categories ***Role*** *(who the LLM should be, i.e a cook)*, ***Context*** *(what it has access to, i.e. what ingredients)*, ***Constraints*** *(what it should pay attention to i.e. a healthy, low fat diet)* and ***Instruction*** *(what output it should produce, i.e. a recipe)*. This is called ***prompt engineering***. By providing information not present at training time, the LLM will generate a ***more specific/reliable output***. An LLMs process of responding to prompts using provided data or context is called ***augmented generation***.

Retrieval Augmented Generation (RAG): Let the LLM ***access a data source*** by uploading files or writing prompts that hint the LLM to send a query to a search engine. Also called ***context-aware LLM***. Use cases: Searching internal, new *(post-training)* or domain-specific data. A popular RAG architecture is LlamaIndex.

2.2. OPEN SOURCE LLMS

Many open source models are available. Two popular models are ***LLaMA2*** by Meta and ***Mistral***. Models are basically defined by two files: ***model.py*** *(small code file that defines the structure of the LLM and runs it)* and ***weights.pkl*** *(potentially huge file with the parameters. Sometimes called a "checkpoint")*.

2.3. WORKING WITH YOUR OWN LLM

Enhances ***security***, protects private/internal data. Depending on use-case: ***Lower costs***, develop a ***domain-expert model*** on your own data. You can do this by ***fine-tuning*** a pre-trained model *(expensive training, new data requires new fine-tuning)* or with ***RAG***.

2.4. MULTIMODAL AI

Modality: The way something is expressed or perceived *(human senses, images, videos, sound)*. ***Multimodal:*** involving several ways of operating or dealing with something. A multimodal AI integrates different modes of data. Internally, different modes are mapped onto similar representations *(image, text, voice representing a tree)*.

3. TECHNOLOGY ETHICS

Problems of AI: Energy consumption, Few players controlling huge market, Conflict of Interest, Deskilling of moral decisions
DIKW Pyramid: ***Data*** *(Discrete, objective facts about an event)*, ***Information*** *(Data with analyzed relationships and connections)*, ***Knowledge*** *(Contextualized Information)*, ***Wisdom*** *(This is the top of the pyramid)*. These four layers lead to real-world decision making.

3.1. TECHNOLOGY ETHICS

Application of ethical thinking to the practical concerns of technology. Take actions that are voluntarily constrained by our judgement – our ethics.

3.1.1. 16 Challenges and Opportunities

1. Technical Safety
2. Transparency and Privacy
3. Beneficial Use & Capacity for Good
4. Malicious Use & Capacity for Evil
5. Bias in Data, Training Sets, etc.
6. Unemployment / Lack of Purpose
7. Growing Inequality
8. Environmental Effects
9. Automating Ethics
10. Moral Deskilling & Debility
11. AI Consciousness, "Robot Rights"
12. AGI and Superintelligence
13. Dependency on AI
14. AI-powered Addiction
15. Isolation and Loneliness
16. Effects on the Human Spirit

3.2. 6 ETHICAL LENSES

The Rights Lens *(Focuses on protecting individual moral rights based on human dignity and autonomy)*, ***The Justice Lens*** *(Ensures fair and equal treatment based on merit or need, addressing various types of justice.)*, ***The Utilitarian Lens*** *(Evaluates actions by their consequences, aiming for the greatest good for the most people.)* ***The Common Good Lens*** *(Promotes actions that contribute to societal welfare and the well-being of all community members)*, ***The Virtue Lens*** *(Emphasizes actions aligned with ideal virtues fostering personal development and character)* and ***The Care Ethics Lens*** *(Prioritizes empathy and compassion in relationships, considering stakeholders' feelings and contexts)*.

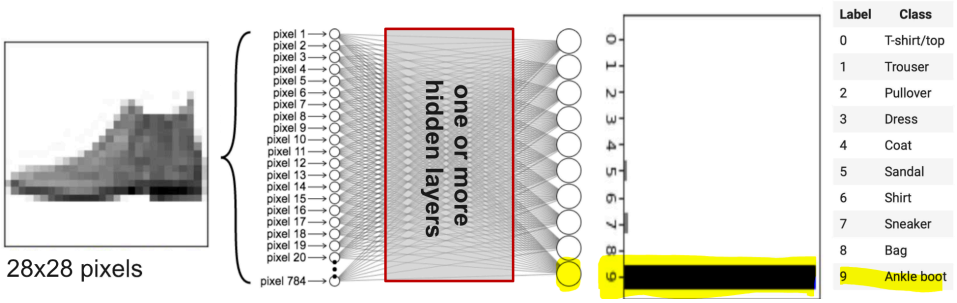
Problems in using the lenses: We may ***not agree*** on the ***content*** of these lenses, not everyone has the same set of civil rights for example. ***Different lenses*** may lead to ***different answers*** to the question "What is ethical?".
Framework for Ethical Decision Making: ***Identify the Ethical Issues:*** *(Determine if the decision could harm or unfairly benefit someone or a group)*, ***Get the Facts*** *(relevant information, stakeholders, explore possible actions)*, ***Evaluate Alternative Actions*** *(Use the lenses for this)*, ***Choose an option for action and test it*** *(select best option, plan carefully)* and finally ***Implement your decision and reflect on the outcome*** *(review results, identify follow-up actions)*

4. IMAGE CLASSIFICATION

Image classification is one of the main tasks of ***supervised learning***, where training data consisting of an image with a label of its content is used for training. The trained model can then predict the most likely label when given an image. It requires 4 things: ***Data***, a ***loss function*** *(typically a probabilistic loss like `SparseCategoricalCrossentropy`)*, a ***model*** and an ***optimizer*** *(`Adam`, an improved version of Stochastic Gradient Descent)*

4.1. EXAMPLE ARCHITECTURE

- ***Input:*** The pixel values of a 2D image are "flattened" into a column vector $(1 \times n$ *vector)* and fed to the hidden layers of an ANN.
- ***Output:*** The ANN has one output neuron per class. Each neuron "votes" for its class. The more active a neuron is, the more likely the input belongs to this class.



4.2. TERMINOLOGY

- ***The Convolution Operation:*** Involves applying a filter (kernel) to input data to create a feature map, detecting patterns like edges in images.
- ***Logits:*** A vector of non-normalized predictions that a classification model generates. Typically passed to a softmax function to generate percentages.
- ***Kernel, Filter, Receptive Field:*** Kernels are small matrices used to apply filters across the input. The receptive field refers to the area of the input that influences a particular feature map value.

- ***Feature Map:*** Result of applying filters to the input, highlighting specific features such as edges or textures.
- ***Tensor, Rank, Dimension, Size:*** Tensors are multi-dimensional arrays. Rank is the number of dimensions, size refers to the total number of elements, and dimensions specify the shape.
- ***Fully Connected Layer (Dense Layer):*** Every neuron in one layer connects to every neuron in the next, crucial for decision-making based on extracted features.

4.3. IMPLEMENTATION

We do not implement a complex ANN from scratch. Instead we use ***TensorFlow***. This is an open source platform for machine learning, which is developed and maintained by Google. We do not use TF directly, instead we use ***Keras***, which provides a higher-level Python API that hides the complexity of TF.

4.3.1. A fully connected ANN with 1 hidden layer

Layers extract representations from the data fed into them.

```
model = tf.keras.Sequential([
  tf.keras.layers.Flatten(input_shape=(28,28)),
  tf.keras.layers.Dense(128, activation='relu'), # Rectified Linear Unit sets negative value to 0
  tf.keras.layers.Dense(10)
])
```

- ***First Layer:*** `tf.keras.layers.Flatten`***:*** Transforms images from 2D (28x28 pixels) to 1D (784 pixels), unstacks rows of pixels and lines them up. No parameters, only reformats data.
- ***Subsequent Layers:*** `tf.keras.layers.Dense`***:*** **First Dense Layer**: Contains 128 fully connected neurons.
  Second Dense Layer: Returns a logits array with a length of 10. Each node indicates the score for one of the 10 classes.

Compiling

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

- ***Optimizer:*** This is how the model is updated based on the data it sees and its loss function.
- ***Loss function:*** This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- ***Metrics:*** Used to monitor the training and testing steps. The example uses accuracy, the fraction of the images that are correctly classified.

Training

```
model.fit(train_images, train_labels, epochs=10)
```

Training the neural network model requires the following steps:

1. ***Feed the training data to the model.*** In this example, the training data is in the `train_images` and `train_labels` arrays.
2. ***The model learns*** to associate images and labels.
3. You ask the model to ***make predictions*** about a test set—in this example, the `test_images` array.
4. ***Verify*** that the predictions match the labels from the `test_labels` array.

Evaluation

```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

Compare how the model performs on the test dataset.

Make predictions

With the model trained, you can use it to ***make predictions*** about some images. Attach a ***softmax layer*** to convert the model's linear outputs (logits) to probabilities, which should be easier to interpret.

```
probability_model = tf.keras.Sequential([model, tf.keras.layers.Softmax()]);
predictions = probability_model.predict(test_images)
```

4.4. TENSOR TERMINOLOGY

- ***Tensor:*** A multi-dimensional array with a uniform data type.
- ***Rank:*** The number of dimensions *(axes)* in a tensor.
- ***Dimension:*** A particular dimension of a tensor, such as rows or columns.
- ***Shape:*** The size of each dimension in a tensor.
- ***Size:*** The total number of elements in a tensor, product of the shape's elements.
- ***Indexing:*** Accessing tensor elements using their indices.
- ***Reshaping:*** Changing the shape of a tensor without altering its data.
- ***Broadcasting:*** Extending tensors of different shapes for arithmetic operations *(scalar product)*.

5. CONVOLUTIONAL NEURAL NETWORK (CNN)

A CNN is an ANN with ***multiple convolutional layers stacked*** (and usually combined with fully connected layers). It is used to process and classify images.
To flatten an image into a long vector is not optimal for processing, because ***neighborhood information*** is lost. Instead, we can apply something called ***convolution***.
In theory, a fully connected network could also learn to classify images an outperform a CNN. The reason is that a ***dense network*** has a ***lower bias but higher variance***. The pre-wired structure of the convolution imposes a ***bias*** to the mode. This ***helps*** the model to learn ***faster***. A model that integrates some domain knowledge into its pre-wired structure requires less data to be trained. This form of bias is called ***inductive bias***.

5.1. FEATURE DETECTOR

For example, one part of the brain detects faces, another objects, another shapes and another lines. Together, these informations form an image.

5.1.1. Mathematical Model

- ***The Input:*** An image *(width × height (× 3 if RGB))* and a filter *(kernel)*, which consists of an $m \times n$ matrix in the simplest case *(1 channel, gray scale)* or an $m \times n \times d$ "stack of matrices" where the depth of the kernel is equal to the number of input channels *(3 for RGB)*.
- ***The operation:*** Convolution of the image with the kernel. The kernel "slides" over the image and on each slide, a matrix multiplication is performed. This can be done in parallel. Multiple kernels are trained to extract different features. Multi-channel inputs need at least as many kernels as channels.
- ***The output:*** This function returns a ***feature map***. This is a "map" *(like a gray scale image)* that shows the locations where a feature is present. One convolution produces one feature map. Even if the input and the filter have multiple channels, the output of the convolution has one channel.

5.2. KERNELS

Kernels *(or filters)* are ***small matrices***. They slide across the input data and perform element-wise multiplication. This ***produces a feature map*** that ***highlights specific patterns*** or features *(like vertical edges)*. The kernel weights are initialized with random data. In the optimization process, they are adjusted so that the loss is minimized. The amount by how many pixels is shifted with each slide is called ***stride***. Usually the same in both dimensions, but can be different. Edges, corners and strides that don't cover the entire input can be dealt with ***padding***: ***"valid"*** *(No padding)* or ***"same"*** *(evenly padded with zeroes horizontally & vertically)*.

5.3. FEEDING FEATURE MAPS INTO THE NEXT CONV-LAYER

The output of a convolutional layer looks like a ***new image***. We can ***apply multiple kernels*** to the same input. This produces ***multiple feature maps*** which can be ***stacked*** *(like an RGB image is stacked from 3 Channels)*. This output can be fed into the next convolutional layer to ***find more complex patterns***.

5.4. POOLING

Instead of directly feeding the next layer, a ***pooling layer*** is usually used. In TF, the default pool size is 2x2. ***Max pooling*** only keeps the largest value in the pool. ***Average Pooling*** keeps the average value within the pool. Comes with a loss of information *(loses details, but keeps important features)*. These layers don't have trainable parameters. ***Advantages:*** Reduces the amount of data to process by the pool size *(2x2 pool → 4x less data)*, Field-of-view increases *(pooled matrix contains information from a wider input)*.

5.5. CNN IN KERAS

```
model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),  # Normalize pixel values
  layers.Conv2D(16, 3, padding='same', activation='relu'),  # 16 neurons, 3x3 kernel, ReLU
  layers.MaxPooling2D(),  # Max pooling layer to reduce spatial dimensions
  layers.Conv2D(32, 3, padding='same', activation='relu'),  # 32 neurons, 3x3 kernel, ReLU
  layers.MaxPooling2D(),  # Max pooling layer to reduce spatial dimensions
  layers.Conv2D(64, 3, padding='same', activation='relu'),  # 64 neurons, 3x3 kernel, ReLU
  layers.MaxPooling2D(),  # Max pooling layer to reduce spatial dimensions
  layers.Flatten(),  # Flatten the tensor to a 1D vector
  layers.Dense(128, activation='relu'),  # Fully connected layer with 128 neurons, ReLU activation
  layers.Dense(num_classes)  # Output layer with units equal to the number of classes
])
```

5.6. CALCULATION EXAMPLE

Image (grayscale)

23	255	40	12	4
21	34	200	43	200
160	180	17	190	80
210	190	40	3	240

Kernel 1

1	0
0	1

bias = 0

Kernel 2

0	1
1	0

bias = -1

Kernel 3

0.9	1.1
-2.8	0.8

bias = 0.15

Kernel 4

0.9	1.1
0	0.8

bias = -1.6

5.6.1. Convolve the input image with kernel 1

Kernel

1	0
0	1

Feature map

57	455	83	212
350	220	20	430

Horizontal stride = 1, vertical stride = 2, Padding: none (valid), Pool: 2x2

Stride *by how many pixels the kernel is shifted.*

$23 * 1 + 255 * 0 + 21 * 0 + 34 * 1 = 57$ *(shift kernel 1 to the right)*

$255 * 1 + 40 * 0 + 34 * 0 + 200 * 1 = 455$

... (shift 2 down when kernel has reached the right edge)

Size of the resulting feature map: 2×4

If a kernel has ***multiple channels***, the channels need to be calculated separately and then added together with the bias to get the final value.

Example: Channel 1 returns 158, Channel 2 returns -14, Channel 3 653. Bias is 1. $\Rightarrow 158 + -14 + 653 + 1 = \underline{798}$

5.6.2. Apply the max pooling operation

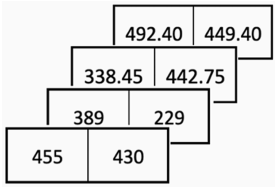
feature map after max pooling

455	430

Using a 2×2 kernel-size, stride = kernel-size (default). This means get the ***max value*** of every 2×2 square.

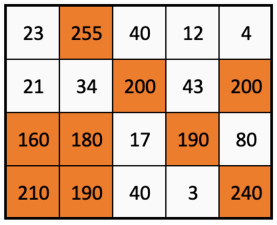
Size of the feature map after pooling: 1×2

5.6.3. Shape of output of one Conv-Layer



After all Kernels are applied, what will be the shape of the output of one Conv-Layer with these four kernels, followed by max pooling? ***Output tensor of rank=3 with shape= (1,2,4)***

5.6.4. Observations



- Applying a threshold ***highlights the structure*** of the "image". ***The different kernels detect different structures***. For example, the "long diagonal" *(255, 200, 190, 240)* is captured by the first kernel. The diagonal 190, 200 is **not** detected by kernel 2 because of the stride of 2.
- By applying ***max pooling*** on these small feature maps, we ***lose many details***.
- Kernel 3 is more specific than kernel 4. ***Kernel 3 is a "corner detector".*** All other kernels, which have 0s, strongly respond to the block in the lower left corner.

5.6.5. Calculation of trainable parameters (degrees of freedom)

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), strides=2, padding='same', activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10))
```

Layer 1, Conv2D

The number of weights/trainable parameters in this layer is calculated as

$(\text{kernel size} \cdot \text{input channels} + \text{bias}) \cdot \text{amount of kernels in this layer} \Rightarrow (3 \cdot 3) \cdot 3 + 1 \cdot 32 = 896$

Note that the number of parameters in a ConvLayer does ***not*** *depend on the width and height of the input*

Output shape: $\frac{\text{input\_width} - \text{kernel\_width}}{\text{stride}} + 1 \Rightarrow \frac{32-3}{1} + 1 = 30$

Without padding, we can apply the kernel 30 times over a width of 32px *(same for height)*. Each of the 32 kernels outputs one 30×30 feature map. Therefore, the output has the shape $\underline{30 \times 30 \times 32}$.

Default values are `strides=(1,1), padding="valid"`

Layer 2, MaxPooling

MaxPooling Layers do not have weights and therefore no trainable parameters.

Output shape: $\frac{\text{input\_width} - \text{pool\_width}}{\text{stride}} + 1 \Rightarrow \frac{30-2}{2} + 1 = \underline{15}$, shape is therefore $\underline{15 \times 15 \times 32}$

Default values are `strides=None`, `padding="valid"`. When `strides=None`, it will default to `pool_size`.

Layer 3, Conv2D

Number of ***trainable parameters:*** $(3 \cdot 3) \cdot 32 + 1 \cdot 64 = \underline{18'496}$

Output shape: Note the `padding='same'` hyper-parameter. This adds rows and columns of 0s to the input, evenly to the left/right or up/down when `'same'` so that the output feature map will have the ***same*** dimensions as the input tensor.

Calculation: The input size is ***increased by 1*** at the left and at the right: $(+(2 \cdot 1)$

$(15 + (+(2 \cdot 1) - 3)/2 - 1 = \underline{8}$. So the resulting shape is $\underline{8 \times 8 \times 64}$

Layer 4, MaxPooling

No trainable parameters.

Output shape: $(8 - 2)/2 + 1 = \underline{4}$, shape is $\underline{4 \times 4 \times 64}$

Layer 5, Flatten

Does not have any trainable parameters.

Output shape: Flatten takes the $4 \times 4 \times 64$ input tensor and reshapes it into a "flat" 1024×1 vector.

Layer 6, Dense

Trainable weights: Each of the 10 neurons is fully connected, plus 1 bias: $10 * (1024 + 1) = \underline{10'250}$

Output shape: 10×1 or simply 10.

Hoy many trainable parameters does the network have?

$896 + 18'496 + 10'250 = \underline{29'642}$ trainable Parameters.

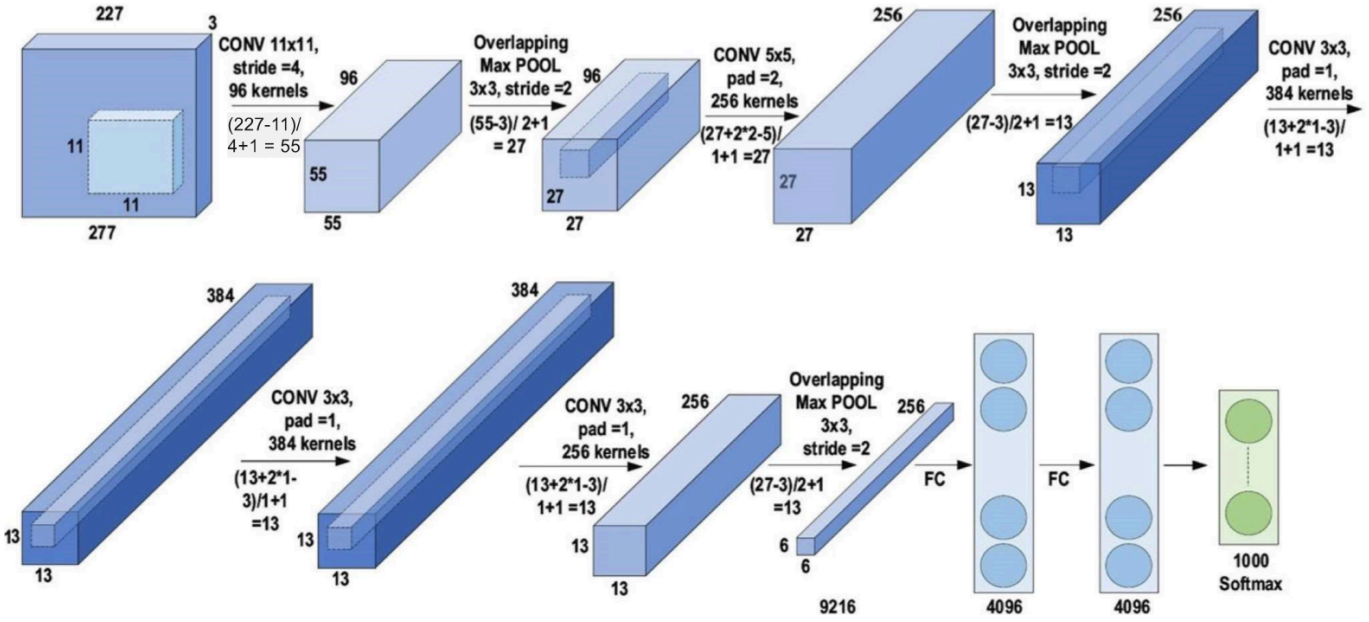
What would happen if the input image is 2-times wider?

The number of trainable parameters in the ConvLayers does not change, but the output feature maps are larger. The Dense Layer would have $\approx 2 \times$ more parameters.

6. DEEP LEARNING ARCHITECTURES

A network is typically called a ***deep neural network*** if it has at least 2 hidden layers.

AlexNet: competed in the ImageNet Large Scale Visual Recognition Challenge in 2012 and ***massively outperformed*** its competition. The original paper's primary result was that ***using Dropout*** in the fully-connected layers was ***very effective***. Even though the model was ***computationally expensive***, it was made feasible due to the utilization of graphics processing units (GPUs) during training.



6.1. SOFTMAX FUNCTION

Softmax ***turns logits*** (numeric output of the last layer) ***into probabilities*** by normalizing each number so the entire output vector adds up to one. A softmax layer is often appended to the last layer of a CNN. Softmax is ***differentiable*** because we need gradient descent to train the model.

$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

6.1.1. Sparse categorical crossentropy Loss

Sparse Categorical Cross Entropy is a loss function that is commonly used. It is an extension of the Cross Entropy loss function that is used for binary classification problems. The sparse categorical cross-entropy is used in cases where the output labels are represented in a sparse matrix format.

- ***Forward Pass (Inference):*** Input passes through the network, the network produces logits which are converted to probabilities. The true label is provided and the loss is calculated.
- ***Backward Pass (Learning):*** Using gradient descent, the network updates its weights in reverse order. The gradients of the loss with respect to each weight are computed. The weights are adjusted to minimize the loss.

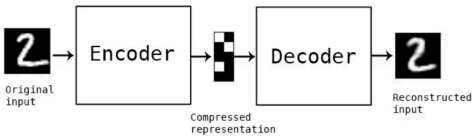
This ***iterative process*** improves the ***model's accuracy*** over time.

6.2. DROPOUT LAYER

A dropout layer is a regularization method used to ***prevent overfitting***. In the dropout layer, a random subset of the current neurons is dropped. `tf.keras.layers.Dropout(0.1)`
The data is often better interpretable ***with*** dropout layers in between because it ***reduces noise*** and allows rapid testing of different "versions" of a model where some neurons are missing. This forces the network to break up situations where neurons have adapted to mistakes from previous layers and thus makes the model ***more robust*** and ***increases its generalization***.

7. AUTOENCODER

An Autoencoder is an unsupervised artificial neural network that learns how to efficiently compress and encode data, then learns how to reconstruct the data back ***from*** the reduced encoded representation ***to*** a representation that is as close to the original input as possible. In other words, it ***reproduces input***. It uses mean squared error as its loss function *(distance between the RGB values of the input & output pixel, sum up all squared distances and divide by number of pixels)*.



It consists of 4 main parts:
- ***Encoder:*** The model learns how to reduce the input and compress the data into an encoded representation.
- ***Bottleneck:*** The layer that contains the compressed representation of the input data. *(latent space)*
- ***Decoder:*** The model learns how to reconstruct the date from the encoded representation.
- ***Reconstruction Loss:*** Method that measures how well the decoder is performing.

7.1. APPLICATIONS

- ***Compression:*** Encoding can group pixels together and therefore be used for compression.
- ***Denoising:*** The bottleneck layer learns useful features which are shared across different images. Isolated/Random pixels (noise) are not representable in the code. Therefore the reconstruction is a composition of the most relevant features, a "denoised" image.

7.2. DECONVOLUTION

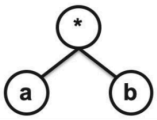
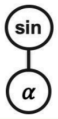
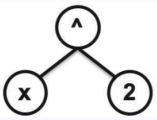
The ***opposite*** process from ***Convolution***. It is also known as Transposed Convolution. In Deconvolution, the ***feature map gets converted into an Image***.

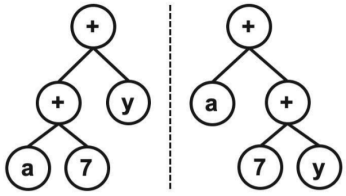
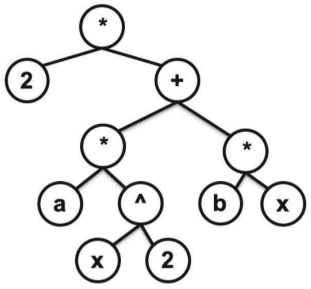
7.3. MEL-SPECTROGRAM

We can also classify audio with an image classifier by transforming the audio into a mel-scaled spectrogram which shows the ***power of different frequencies over the time***. This transformation can be done with Fourier transform.

8. REPRESENTATIONS

8.1. EXPRESSION TREES

Expression (elementary)	Expression Tree
5	
x	
a * b *analogous: + - ** *(binary operations)*	
sin (α) *analogous: cos, tan, inv, sqrt, exp, ...* *(unary operations)*	
x^2	

Expression (composed)	Expression Tree
a + 7 + y *= (a + 7) + y* *or* *= a + (7 + y)*	
2 * (ax^2 + bx) we us the elementary building blocks and combine them to form arbitrarily complex expressions.	

There is not always just one solution, there might be different trees for the same expression.

8.2. BACKPROPAGATION

Backpropagation is a supervised learning technique to ***adjust the weights of the neurons to minimize the error***. It calculates the gradient of the loss function with respect to each weight in the network. This gradient is then used to update the weights in the opposite direction of the gradient, which in turn minimizes the loss function.

The algorithm works by computing the error between the predicted output and the actual output for each example and then ***propagating*** this error back through the network to adjust the weights. This is repeated multiple times until the weights converge to a point where the error is minimized.

Steps of the algorithm:

1. ***Initialize the weights*** of the network ***randomly***
2. ***Forward propagate an input*** through the network to get the predicted output.
3. ***Compute the error***
4. ***Backward propagate the error*** through the network to compute the gradient
5. ***Update the weights*** in the opposite direction of the gradient using SGD or something similar
6. ***Repeat*** steps 2-5 for multiple iterations

8.3. GRADIENT DESCENT

MSE loss function for a quadratic model and a single data point x, y:

$$L(\text{values the function depends on}) = (\text{loss function} - \text{datapoint i})^2 \Rightarrow L(x, y; a, b) = (ax^2 + bx + c - y)^2$$

9. ADVANCES TECHNIQUES IN KERAS

9.1. DATA AUGMENTATION

The ***convolution*** is ***translation-invariant*** *(Object still identifiable if shifted along x or y axis)* but not ***rotation-invariant*** *(CNN fails to classify a rotated image)*. It is also in general not ***scale-invariant*** *(a scaled object does not get recognized)*.

A classifier can only classify data that is ***similar to the training data***. If we have not enough training data, we need to artificially create more by applying different transformations *(De-texturized, de-colorized, edge enhanced, salient edge map, flip, rotate, …)*.

```
layers.Resizing(IMG_SIZE, IMG_SIZE), layers.Rescaling(1./255),
layers.RandomFlip("horizontal_and_vertical"), layers.RandomRotation(0.2),
skimage.color.rgb2gray(imgs)
```

9.2. BATCH NORMALIZATION

Batch normalization applies a transformation that maintains the ***mean output close to 0 and the standard deviation close to 1***. ***Advantages:*** We can train deeper networks and increase the learning rate. This is added as a layer inside the model.
`keras.layers.BatchNormalization( ... )`

9.3. LEARNING RATE SCHEDULING

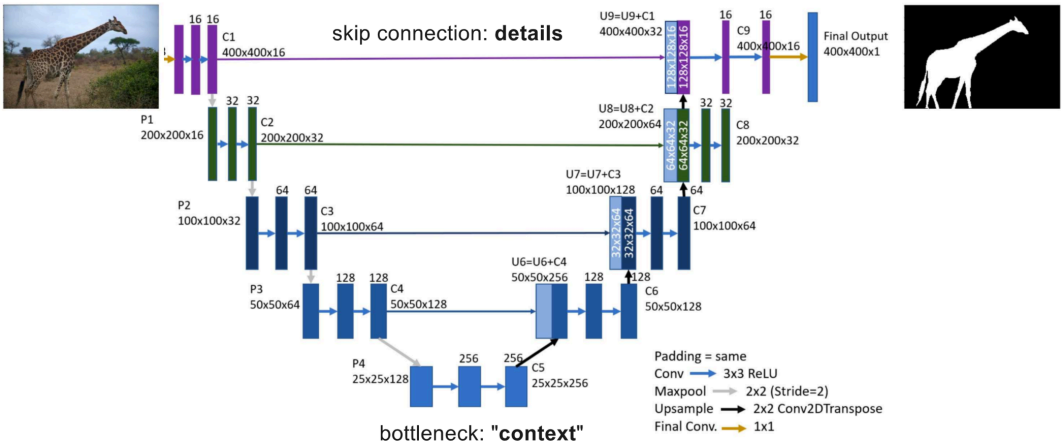
A constant learning rate is often not optimal. It is often necessary to try different learning rates and schedulers. ***Annealing*** is one example of Learning Rate Scheduling.
With a scheduler, we can have ***faster training*** *(higher learning rate in the beginning)* and ***better convergence into a (local) minimum***.
`keras.optimizers.schedules.CosineDecay( ... )`

9.4. ACCURACY VS. LOSS

Accuracy: ***Counting the correct samples.*** All samples contribute the same amount to this value, they are either correct or incorrect. This ***cannot be optimized*** with a function.
Loss: The loss is a ***differentiable*** function which ***can be optimized***. Very confidently wrong predictions make the loss explode.

9.5. U-NET ARCHITECTURE

Is a "classic" architecture for image segmentation that splits the image into fore- and background.



9.5.1. Complex Network Topologies

Skip Connections: Keep a pointer to the output of a layer. Pass that data unchanged to a deeper layer in the network.

```
skip1 = layers.Conv2D(1, kernel_size=1, activation=None, padding='same', strides=1)(x)
... # potentially many more layers here
x = layers.concatenate([x, skip1])
```

Multiple Outputs: An ANN can have multiple output layers. Each output layer can have its own Loss Function.

10. REINFORCMENT LEARNING (RL)

Behavioral Learning: Learning is the ***change in behavior*** that occurs as a ***result of experience***.
In RL, an ***agent interacts with its environment***. The agent ***selects from available actions***. The agent's goal is to select those actions which maximize the (long term) reward. In the beginning, the agent selects ***random actions***. Over time, the agent ***learns*** to ***prefer*** those actions that yield ***higher*** (long term) ***rewards***: ***trial and error***.
RL does ***not learn from "pre-collected" labelled / unlabeled data***. It learns from ***interaction*** with the environment.
Fields of Application: Healthcare, Education, Transportation, Energy, Business Management, Science, NLP, Computer vision, robotics, games, computer systems, finance, ….
Limitations: ***Sample efficieny*** *(No real progress in the first couple million trials)*, ***Difficult*** to define the ***reward function***.

10.1. BASIC CONCEPTS

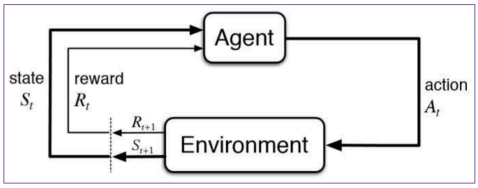
An RL agent tries to solve an RL-problem by learning the actions that maximize reward.
- ***Environment:*** The Environment the agent is in. Contains states and actions.
- ***Agent:*** Knows its state *(but not the entire environment, only "sensor data")*, observes the states, takes actions and learns optimal behavior to maximize reward *(hedonistic behavior)*.
- ***State:*** The agent arrives in a new state after taking a action *(S0, S1, S2 etc.)*.
- ***Terminal State:*** A state where there aren't any further actions that can be taken by the agent *(Terminates an episode)*.
- ***Episode:*** One "run" of the agent through the environment until it lands in a terminal state.
- ***Actions:*** The agent takes an action to get from one state to another *(a0, a1)*.
- ***Rewards:*** The agent gets a positive or negative reward when it arrives in a new state. Through this, the agent learns what "good" and "bad" actions are.
- ***Policy π:*** Fully defines the behavior of the agent. Maps states to action-selection probabilities. Normally an agent starts with a random policy and then tries to improve it. *("At state S53, go left with probability 76% and right with 24%" is noted as $\pi(s = 53, a = \text{left}) = 0.76$)*

10.2. MARKOV DECISION PROCESS (MDP)

Composed of 4 entities: ***S*** *(set of states {S0, S1, … S7})*, ***A*** *(set of actions {a0, a1})*, ***R*** *(positive or negative rewards)* an ***P*** *(Transitions, visualized with arrows)*. MDPs are a very general mathematical function. They are used to describe and study a large variety of problems.

10.3. GENERAL RL FRAMEWORK

An agent ***decides*** which ***action*** to take ***according to its policy*** π. The action has an ***effect on the environment***. As a result, the environment ***transitions to the next state*** and ***returns a reward***. This ***loop*** continues ***infinitely*** or until a ***terminal state*** is reached.



10.3.1. Goal of an RL

"Find the optimal policy π^*" by trial-and-error. π^* is: At each state S_t, take the action A_t which returns the ***largest sum of (discounted) rewards***.

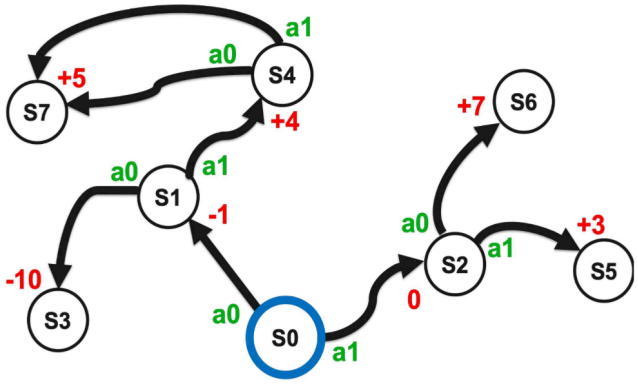
10.3.2. Different policies, different sum of rewards (this is without discounting)

Optimal policy π^*

State	a0	a1
S0	1	0
S1	0	1
S2	1	0
S3	-	-
S4	0.5	0.5
S5	-	-
S6	-	-
S7	-	-

Random policy π^{random}

State	a0	a1
S0	0.5	0.5
S1	0.5	0.5
S2	0.5	0.5
S3	-	-
S4	0.5	0.5
S5	-	-
S6	-	-
S7	-	-



Assume the agent starts in state x and follows π^*. Total reward?

State	Total Reward
S0	$V^{\pi^*}(\text{S0}) = -1 + 4 + 5 = 8$
S1	$V^{\pi^*}(\text{S1}) = 4 + 5 = 9$
S2	$V^{\pi^*}(\text{S2}) = 7$
S3	$V^{\pi^*}(\text{S3}) = 0$

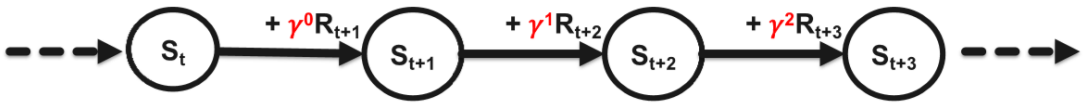
Same for Random Policy. What does the agent collect on average? Work backwards.

State	Total Reward
S7	$V^{\pi^*}(\text{S7}) = 0$
S4	$V^{\pi^*}(\text{S4}) = \pi(\text{S4}, \text{a0}) \cdot 5 + \pi(\text{S4}, \text{a1}) \cdot 5 = 0.5 \cdot 5 + 0.5 \cdot 5 = 5$
S1	$V^{\pi^*}(\text{S1}) = 0.5 \cdot -10 + 0.5 \cdot (4 + 5) = -5 + 4.5 = -0.5$
S0	$V^{\pi^*}(\text{S0}) = 0.5 \cdot (-1 + -0.5) + 0.5 \cdot (5 + 0) = 1.75$

This is the function used for this table: $V(\text{S0}) = \pi(\text{S0}, \text{a0}) \cdot (R + V(\text{S1})) + \pi(\text{S0}, \text{a1}) \cdot (R + V(\text{S1}))$
R: Reward of A, $V(S_1)$: V-State of the state the action leads to. If this is a terminal state, this is zero.

10.3.3. Discounting

A ***far away award is less attractive***. In a discrete-time RL, it is common to apply a ***constant discount factor*** γ at each time-step to steer the agent away from far away rewards towards nearer ones with the same or a similar reward. The value of γ depends on the problem, but typical values are 0.95, 0.98, 0.99, 0.999. Discounting is a simple and efficient strategy to lump together all kinds of risks.



Discounting is only applied on ***future rewards***. In each state, the agent receives the actual reward r. When in state S_t, a reward 3 steps into the future is discounted by the power of 2: $y^2 R_{t+3}$. But when actually moving there, the agent receives r_{t+3}, not $y^2 R_{t+3}$.

Exercise
While in state S0, evaluate the discounted reward of the two paths in the image in Section 10.3.2 for $\gamma = 0.95$.
τ_1: S0-S1-S4-S7 $= -1 \cdot (0.95)^0 + 4 \cdot (0.95)^1 + 5 \cdot (0.95)^2 = 7.3$
τ_2: S0-S2-S6 $= 0 \cdot (0.95)^0 + 7 \cdot (0.95)^1 = 6.65$
The trajectory τ_1 has a ***higher discounted reward*** and gets therefore chosen.

Reward R is the ***quantity an agent receives immediately*** when landing in a state. Usually described with r_t
(r_t is used for the actual value that has been received by the agent, R_t is used for the random variable before observing the actual outcome)
Return G: is the ***discounted sum of future rewards***. Is a sum of random variables and therefore a random variable itself. Usually the symbol G_t is used.

$$G_t \overset{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

10.3.4. State-action value $Q(s, a)$
The state-action value $Q^\pi(s, a)$ is defined as the expected Return G, when starting in state s, taking action a, and following the policy π thereafter.

$$q_\pi(s, a) \overset{\text{def}}{=} \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

expected Return G, Expected return when following policy π, Reward we get when we are in this state

The goal of many RL algorithms is to estimate these q-values. Q-values are expectations and depend on the policy and on γ. To calculate a Q-value, you only need the next state/action values. A simple method to approximate expectations is ***sampling:*** simulate many trajectories, observe the rewards and calculate the mean reward. ***The sample mean is an approximation of the expectation.*** This is called ***Monte-Carlo*** Method.

10.4. TEMPORAL DIFFERENCE LEARNING
In state s_t, taking action a_t, the agent expects the future reward $q(s_t, a_t)$. After taking a step, the agent observes the next state and reward: $q(s_t, a_t) \overset{?}{=} r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$ *($q(s_t, a_t)$ is the expected future reward before taking a step, r is an actual observation and $q(s_{t+1}, a_{t+1})$ is the expected return from here on)*
This equation ***does not hold***. We need to add a ***difference δ_t*** to make both sides equal:

$$q(s_t, a_t) + \delta_t \overset{?}{=} r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$$

δ is needed because of the difference between the expected value and the actual value. *(i.e. the average expected value of a dice roll is 3.5, but the actual value can never be 3.5)* We can now start with any random initial guess for the q-values and then use the ***temporal difference error / reward prediction error (RPE) δ_t*** to ***improve*** our guess. Over time, the q-values will ***converge*** towards the "true" expected values and $\delta = 0$ *(on average)*. Positive RPE means we got more than predicted.

The TD-Learning update rule:
$\text{RPE} = r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)$
$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha \cdot \text{RPE}$ *(updated guess overwrites the old guess)*
α is the ***learning rate*** or ***step size*** *(typically between 0.000001 and 0.1)*.

10.4.1. State-Action-Reward-State-Action (SARSA)
Unlike Monte-Carlo, SARSA integrates ***new information*** at ***every step***, not just at the end of an episode. The agent in state S chooses action A, lands in state S' and gets reward R. It then chooses its next action A' based on its policy Q. The initial guess $Q(S, A)$ gets updated based on the agents observation of S, A, R, S' and its choice of A'. The RPE calculates the difference between the predicted $Q(S, A)$ and the actual value $Q(S', A')$.
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
$\Leftrightarrow$ Immediate reward + discounted [prediction in S', A' − Q-value of prediction when we started]

Assume $\alpha = 0.1$ and $\gamma = 0.9$. An agent performs the following state-action sequence:
S1 - a3 - S3 - a3. **Which entry of the Q-Table gets updated?** The 2.1
When landing in S3, it receives a $+4$ reward. Calculate the updated Q-value with SARSA:
$\text{RPE} = 4 + 0.9 \cdot (-1.4) - 2.1 = \underline{0.64}$
$2.1 \leftarrow 2.1 + 0.1 \cdot 0.64 = \underline{2.164}$

	S1	S2	S3
a1	1.1	-0.9	1.3
a2	0	1.1	1.2
a3	2.1	1.5	-1.4

Q-Learning: A variant of SARSA. The update of $Q(s,a)$ is based on the agent's observation or r and s', and ***the Q-value of "best" action in S'***: $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$

The next action a' can differ from the best action. That means, in the next step, the agent can take one action a', but use another action $\max_a$ for the Q-Update. Algorithms that learn from actions that differ from the actual taken action are called ***off-policy***.
SARSA = on-policy *(takes this action in the next step)*, Q-learning = off-policy *(next step decided by Q-learner based on best Q-value.)*

10.4.2. SARSA and Q-learning Calculations

An agent has interacted with the Treasure-Map environment and approximated the following Q-Table *(state × action)*:

```
array([
  [ 4.0, 5.9],
  [-8.5, 6.5],
  [ 5.0, 2.8],
  [ 0.0, 0.0],
  [ 5.0, 4.8],
  [ 0.0, 0.0],
  [ 0.0, 0.0],
  [ 0.0, 0.0]
])
```

SARSA:
$\text{RPE}_t = r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t), \quad Q^\pi(s_t, a_t) \leftarrow Q^\pi(s,a) + \alpha \text{RPE}_t$
Q-learning:
$\text{RPE}_t = r_{t+1} + \gamma \max_{\tilde{a}} Q^\pi(s_{t+1}, \tilde{a}) - Q^\pi(s_t, a_t), \quad Q^\pi(s_t, a_t) \leftarrow Q^\pi(s,a) + \alpha \text{RPE}_t$
When moving form `S0` to `S2`, the RPE is always 0 because when landing in stat `S2` the reward is 0:
This is wrong because the RPE depends not only on the immediate reward.
The agent is in state `S0` and follows the trajectory $\tau =$ `S0 - a0 - S1 - a0 - S3`. Assume $\gamma = 0.95$ and $\alpha = 0.05$

Calculate the RPE and the updated $Q(\text{s0}, \text{a0})$:
$Q(\text{s0}, \text{a0}) = 4, \quad r = -1$ *(see image in Section 10.3.2)*
$Q(\text{s1}, \text{a0}) = \text{-8.5} \Rightarrow \gamma Q(\text{s1}, \text{a0}) = 0.95 \cdot \text{-8.5} = \text{-8.075} \Rightarrow r + \gamma Q(\text{s1}, \text{a0}) = \text{-1} + \text{-8.075} = \text{-9.075}$
RPE SARSA: $r + \gamma Q(\text{s1}, \text{a0}) - Q(\text{s0}, \text{a0}) = \text{-1} + \text{-8.075} - 4 = \text{-13.075}$
$Q(\text{s0}, \text{a0})$ **after SARSA update:** $Q(\text{s0}, \text{a0}) + \alpha(\text{RPE}) = 4 + 0.05 \cdot \text{-13.075} = \underline{3.34625}$
$\max_{\tilde{a}} Q(\text{s1}, a) = 6.5 \Rightarrow \gamma \max_{\tilde{a}} Q(\text{s1}, \tilde{a}) = 0.95 \cdot 6.5 = 6.175$
$\Rightarrow r + \gamma \max_{\tilde{a}} Q(\text{s1}, \tilde{a}) = \text{-1} + 6.175 = 5.175$
RPE Q: $r + \gamma \max_{\tilde{a}} Q(\text{s1}, \tilde{a}) - Q(\text{s0}, \text{a0}) = \text{-1} + 6.175 - 4 = 1.175$
$Q(\text{s0}, \text{a0})$ **after Q-Value update:** $Q(\text{s0}, \text{a0}) + \alpha(\text{RPE}) = 4 + 0.05 \cdot 1.175 = \underline{4.05875}$

10.4.3. Epsilon-Greedy-Policy

In order to discover the best available options, an agent needs to ***explore*** the ***entire*** state-action-space. In contrast, in order to ***maximize the return***, the agent wants to follow the ***best trajectory***. Therefore, we need a way to describe a ***flexible behavior***. If the agent just follows the ***most promising path*** based on its learned knowledge, he ***stops exploring*** and ***starts exploiting***, thus always taking the "best" known action: a ***greedy policy***. This way he might ***miss the best trajectory***. We need to ***balance*** between exploration and exploitation with the epsilon-greedy-policy:

```
at each state:
  with probability 1-ε: Take the action that has the highest value (greedy exploit)
  with probability ε: Take a random action (explore)
```

Finding a good value for ε is key. Possible strategies: fixed at for example 0.1 or starting at 0.95 *(high exploration)* and reducing/ ***annealing*** it over time to 0.05 *(greedy). $\varepsilon = 1$ means random behavior.*

11. DEEP REINFORCEMENT LEARNING

Disadvantages of Q-tables: The Q-Table maps (s,a) *(state-action pairs)* to $Q(s,a)$ *(expected reward of (s,a))*. Since it is a table, it does ***not scale well***. It is ***limited to discrete states and discrete actions***. There is also ***lack of generalization***, there is no relation to neighboring states and actions. In real life, nearby/similar states often have similar Q-Values and it is reasonable to apply similar actions. While the input is often ***high-dimensional***, the relevant features are often not *(background vs. relevant part of an image)*. A tabular-RL would classify each changed pixel as a completely independent state *(senseless!)*.
Since Q is a function, we can ***approximate*** it. In DRL, the Q-values are approximated using a Deep Neural Network. ***Input:*** the state s *(e.g. RGB image, state of a chess board)*, ***Output:*** one neuron per action a. The neurons activity is $\approx Q(s,a)$.

11.1. DEEP Q-NETWORK (DQN)

Replay buffer: The transitions that an agent performs during an episode are stored in the replay buffer as a ***experience***. Randomly pick a few experiences, do a forward- and backward-pass to ***train*** the network on this ***batch*** of samples.
Target Network: This is a network that is updated ***more slowly*** compared to the Q-Network, which is used to define the optimal policy based on the value estimation. The target network is ***not*** updated every iteration but once in a while to ***stabilize*** the learning process.
Epsilon-greedy Policy: Given Q-values, which actions should be taken? Epsilon-greedy is simple to implement and often works well enough for a DQN. If two actions are ***almost equally good***, it is better to use Softmax.

Softmax: T or τ is the ***temperature***. It gives us control over the ***exploration / exploitation*** balance. ***High τ:*** all actions have almost the same probability *(Even with big differences between the Q-Values, they have a similar probability)*. ***Low τ:*** for $\tau \to 0$, the policy becomes greedy. Often, τ is annealed *(becomes more greedy over time)*.

12. SEQUENTIAL AND TIME SERIES DATA

Sequential Data: Words, sentences, streams of text, etc.
Time-Series data: Is a ***subset*** of sequential data where the x-axis is time *(Stock option pricing, Sensor Data, etc.)*.
Sequential Data is everywhere. It is a tremendous source of information. It is also a ***indispensable tool in science*** and there are a lot of ***business opportunities*** and ***applications*** *(Predictive maintenance, portfolio management, sales, politics, weather prediction, forecasting traffic, text generation, anomaly detection, …)*.
Model-based Prediction: Based on physics and historical data, ***mathematical models*** are developed. The sequential data provide the initial conditions. Then the models are used to calculate the evolution. The ***quality*** depends on many factors *(quality of measurements, quality of the model, nature of the dynamic system)*. For most time-series, we ***do not have a causal model***. Therefore, predictions need to be made ***based on historical data***. This is done by looking for ***structure/patterns*** *(Seasonality, Trends)*.
Feature Engineeering: Assuming some function and optimize the free parameters.
Learning from Data: Make very few assumptions about the structure (low bias), machine learning algorithms ***discover*** structure from the data. This can ***outperform*** model-based predictions, for example in the weather forecast.

13. RECURRENT NEURAL NETWORK (RNN)

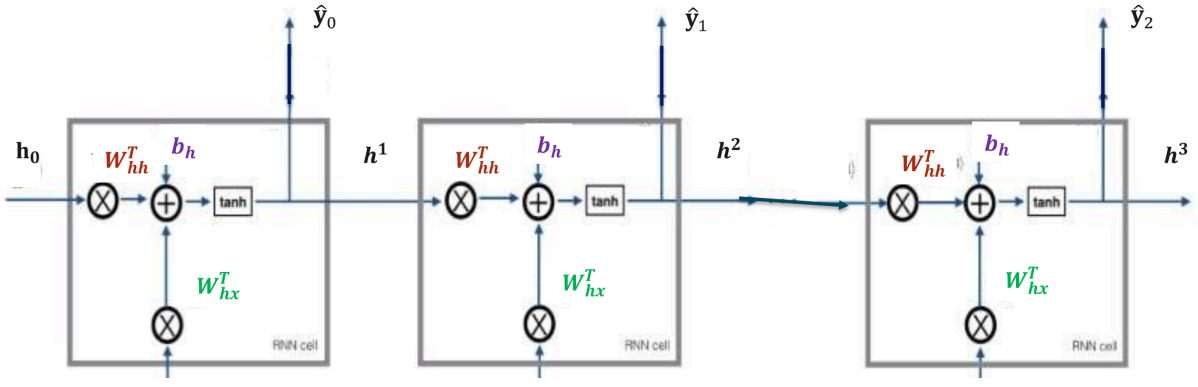
13.1. MODELS

One-to-one *(Sequence has only one time step (static). One input is fed and one output is generated. This is the case in traditional ANNs.)*, ***Many-to-one*** *(sentiment classification - positive/negative sentence)*, ***One-to-Many*** *(Image Captioning)*, ***Many-to-many*** *(Machine Translation)* ***ANN*** and ***FFNN*** *(Feed Forward Neural Networks)* are no good match for sequential data because they are ***unable*** to ***capture the temporal order*** of a time series, since they treat each input ***independently*** *(results aren't based on past data)*. Ignoring the temporal order restrains performance. ***CNN*** is partially good for this task, you can use ***filters*** and flatten the input features to 1 dimension (time). If the forecast depends on ***detecting specific patterns***, then ***CNNs are a good fit***. However, convolutions and pooling operations ***lose information about the local order of words***. The meaning could be misinterpreted or the grammar could be incorrect if sequential information is not used.

13.2. RNN CONCEPT

RNNs introduce a ***recurrent connection*** which allows information to flow from one time-step to the next. This allows the RNNs to ***maintain internal memory*** and utilize information from the previous step for the current step and therefore ***learn temporal dependencies***.

13.3. ARCHITECTURE



Note, this is the same cell, but unfolded through time.
Input: x_t, could be a one-hot vector corresponding to a word.
Hidden state: h_t represents a hidden state at a time t and acts as "memory" of the network. It is calculated based on the current input and the previous time step's hidden state: $h_t = f(x_t, h_{t-1})$. f is parameterized by ***weights and bias*** which need to be ***learned***. These are ***shared*** across all timestamps.
Weights: The inputs x_t and h_t are multiplied with their weights W_{hx}^T and W_{hh}^T respectively.
Bias: With an addition, b_h is added to the output of h_t and x_t after their respective weights have been added.
Activation function: The biased value is put through g, a nonlinear activation function $h_t = g(W_{hh}^T h_{t-1} + W_{hx}^T x_t + b_h)$
Output: The model returns $\hat{y}$, the prediction for this timestep. This value is fed into the network again under the name h_t.

13.4. TRAINING

Backpropagation: Initialize weights, and repeat until convergence: Forward pass, calculate loss, calculate gradient, update weights in backward pass.
Backpropagation through time (BPTT): The error is propagated backward through time until the initial timestep.
Loss Function: ***Binary cross entropy*** for binary classification, ***categorical cross entropy*** for multi-class classification. For regression, use ***RMSE***.
Keras:
```
model = Sequential() # 3 timesteps, 1 feature, 32 neurons
model.add(SimpleRNN(units=32, input_shape=(3,1), activation = "tanh"))
```

13.5. LIMITATIONS

Exploding Gradient: As the backpropagation algorithm advances backwards, the gradient can get ***larger and larger*** *(huge weight changes)* and therefore ***never converges*** on the optimum. ***Gradient Clipping*** and ***proper weight initialization*** can help.
Vanishing Gradient: As the algorithm advances backwards, the gradient can get ***smaller and smaller*** and approach zero, therefore ***never converges*** on the optimum. It is harder and harder to propagate errors from the loss back to distant past. As a result, it might just learn ***short term dependencies***. Fix with ***batch normalization***, ***Long short-term memory***, ***weight initialization***.

14. TRANSFORMERS AND THE ATTENTION MECHANISM

Transformers are the ***most important deep learning architecture*** for sequence modeling. They make use of a powerful building block: ***the attention mechanism***.
Limitations of (simple) RNNs: All information about past patterns in the sequence must be conveyed through the hidden signal h *(context vector)*. The longer the sequence, the more information is "faded out". h is a ***bottleneck***.
Techniques to overcome these limitations: ***Long Short-Term Memory (LSTM)*** has an additional "conveyor belt" signal c. During training, the LSTM learns when to keep / use / modify this signal. This makes an LSTM a ***trainable memory unit***. LSTM still fail to learn from very long sequences.
We can't feed an ***entire sequence*** into a fully connected network: Variable length, input is treated ***independently***, ***exploding*** no. of parameters. We need to give this "clueless" network a clue: ***Inductive Bias*** aka ***attention***.

14.1. ATTENTION

Attention assigns varying ***levels of importance*** to different words in a sentence, by calculating "soft" weights *(its embedding)*, within a specific section of the sentence called the ***context window*** to determine its ***importance***. Attention "looks" at all words *(all tokens)* in the input sequence. Each attention head has its own weights W_q, W_k, W_v. Therefore, different heads look for different patterns, but all look at all words.

14.2. PREPROCESSING

Tokenization: Long text strings are broken down into ***short chunks*** *(1 token = 1 word or a part of a word)*. The ***output*** is a ***sequence of integers***.
Embedding: The word (or token) indices are then ***mapped*** onto a ***vector*** with a ***position encoding***. In Keras, this is implemented using a special type of layer: `keras.layers.Embedding`
Positional Encoding: For text processing, the ***relative position*** of each word ***within the sequence is relevant*** *(sentence structure)*. For each word, a ***"location vector"*** is calculated and added (or concatenated) to its embedding vector before it is fed into the encoder. A common technique is based on ***trigonometric functions*** sine and cosine using ***different frequencies***.

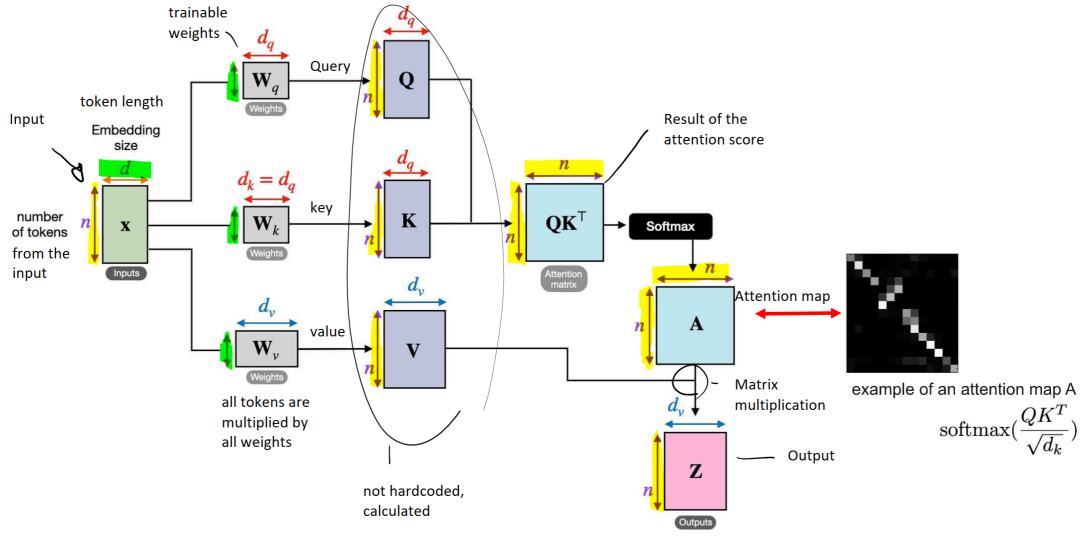
14.3. THE TWO COMPONENTS OF TRANSFORMERS

Encoders: A stack of ***Encoders*** maps the input onto ***embedding vectors***. Encoders can process the entire input sequence in parallel. The sequence of input vectors is passed through a ***self-attention layer***. For each vector in the input, it produces a ***weighted*** linear combination of all input vectors. The same (fully connected) ***feed forward network*** is then applied to each vector. This layer performs a nonlinear transformation. The output is fed into the next encoded layer.

Decoders: A stack of ***Decoders*** map the embeddings to the ***desired target sequence***. The Output generation is (usually) word-by-word. Each decoder consists of a ***self-attention layer***, an ***encoder-decoder attention layer*** which combines the vectors from the decoder below, with the embeddings from the encoder, and a ***feed forward network***.

14.4. CALCULATION OF ATTENTION

The ***attention head*** consists of the matrices ***query*** Q, ***key*** K and ***value*** V. The goal of ***attention*** is to look-up the ***value*** V of those "neighbors" which are strongly influencing the current token *(words with similar meanings)*. An ***attention-score*** is calculated between each pair of the embedding vectors. The attention mechanism is a "softmax-version" of a dictionary: The ***attention score*** is a ***similarity measure*** between a ***key*** K and a ***query*** Q. Both are vectors, the dot-product is used to calculate how strongly they match. ***Strong match***: *(cosine similarity close to 1)* "most" of the value V is returned *($V *$ cosine similarity)*. ***No strong match:*** a close to 0 vector is returned. ***The result of this softmax-lookup is the sum of all values, weighted according to the key/value similarity.***



More compact representation: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ *(The $\sqrt{d_k}$ stabilizes the calculation)*

15. TRANSFER LEARNING, FOUNDATION MODELS

Transfer Learning: Train a complex model on Task A, then freeze part of the model and ***reuse it for a new / related task B***. Useful when you have only a few labels / data for your actual task or if you just don't want to do all of the training by yourself.
Foundation Model: A large model trained on a vast quantity of data at scale, resulting in a model that can be ***adapted*** to a ***wide range of downstream tasks***.
In-Context Learning: The LM performs a task just by ***conditioning on input-output examples***, without optimizing any parameters. It has ***no training phase***, the examples provided in the augmented prompt are ***directly influencing*** the generative process.

16. MACHINE LEARNING OPERATIONS (ML-OPS)

Components of real-world AI projects: Data, Versioning, CI/CD, Monitoring, Reproducibility, Time/Budget constraints, …
CI / CD / CD: Continuous integration, Continuous delivery, Continuous Deployment.
Training pipeline: ***Data Collection*** *(Provided by customer, generated using models or collected from company experts)*, ***Data Preparation*** *(augmentation, sampling)*, ***Model (Re-)Training*** *(infrastructure, consistent versioning, optimization)*, ***Model Evaluation*** *(acceptance testing, performance evaluation, reproducibility)*, ***Model Deployment*** *(manage rollout of different networks, track and document)*, ***Model Monitoring*** *(monitor performance, identify problematic cases, collect feedback and re-train)*.
Data Engineer: Understands ***general workflows*** and data flows with all specific requirements.
DevOps: Has a ***technical skill set***: Operate and maintain processing resources, network infrastructure.
Data Analyst, ML Engineer: Has ***in-depth knowledge*** of underlying ***algorithm*** and processes.

16.1. PROBLEMS WITH DATA

Biased data *(unrepresentative of all data)*, too many ***similar/redundant samples*** *(car with a road)*, ***missing edge cases*** *(pedestrian coming out of nowhere)*, ***bad quality*** *(pictures during a rainy day)*
This can lead to ***bad models***. To improve we need to ***collect more data*** in ***"missing areas"*** or ***balance/remove redundant data***.
The last step can be achieved via ***self-supervised learning*** where similar/redundant pictures are automatically categorized as less relevant. This can decrease dataset size and costs.
When the training data is ***modified***, it should also be ***versioned*** to measure performance across different iterations and to create ***reproducible models***. Can be done via Git.