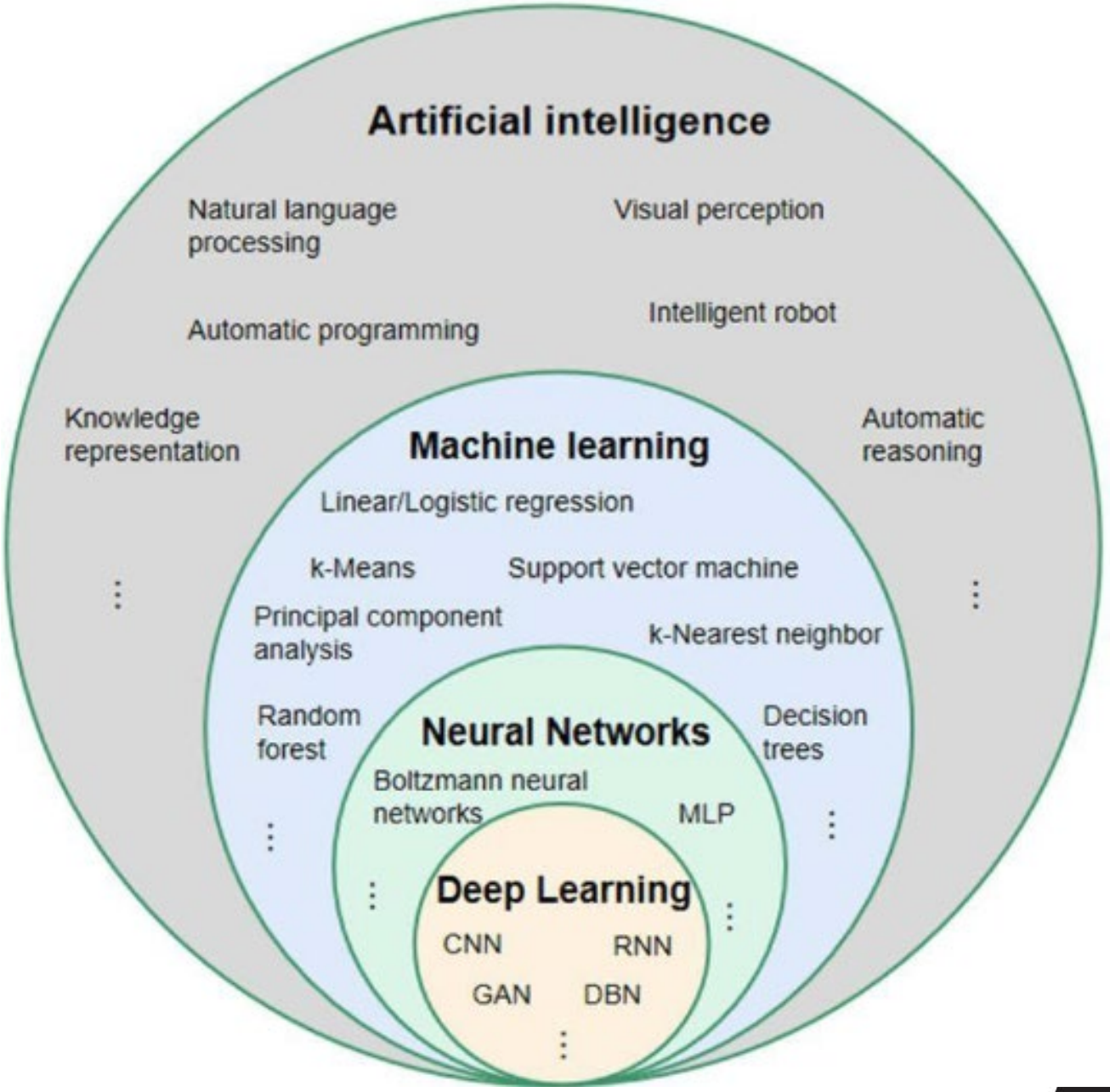
What is Reinforcement Learning (RL)
Elements of RL
Bellman Equation
Q-values

THE UNIVERSITY OF SOUTHERN MISSISSIPPI

What is AI?

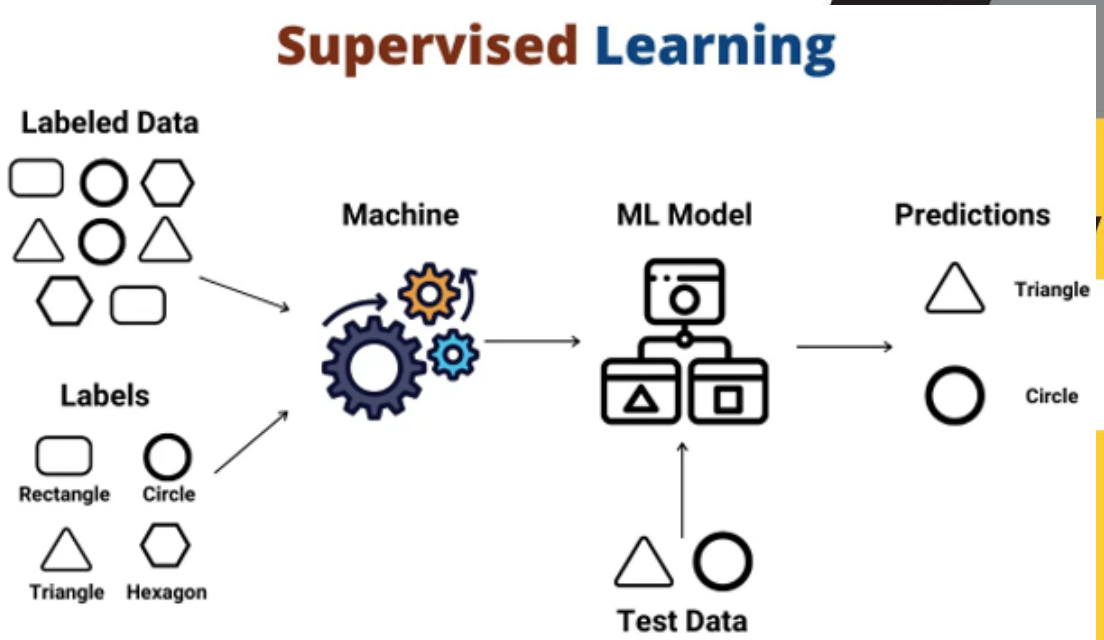

Artificial intelligence
Natural language processing
Visual perception
Automatic programming
Intelligent robot
Knowledge representation
Automatic reasoning
Machine learning
Linear/Logistic regression
k-Means
Support vector machine
Principal component analysis
k-Nearest neighbor
Random forest
Decision trees
Neural Networks
Boltzmann neural networks
MLP
Deep Learning
CNN
RNN
GAN
DBN

What is RL?

- RL is a subfield of Machine Learning.
  - Addresses the problem of automatic learning
    - This is: optimal decisions over time
  - For example:
    - Classifying images (dogs): a program that can classify dog breeds
      - After some time: dog coat style changed; your program cannot classify as well now.
    - How do we address this problem?
      - Add some type of “prediction”. RL incorporates a “prediction” dimension into its equations
      - But, can the prediction be manipulated to compensate for “unforeseen” events in the future?
        - How far can this be manipulated?
        - Is it good or bad?

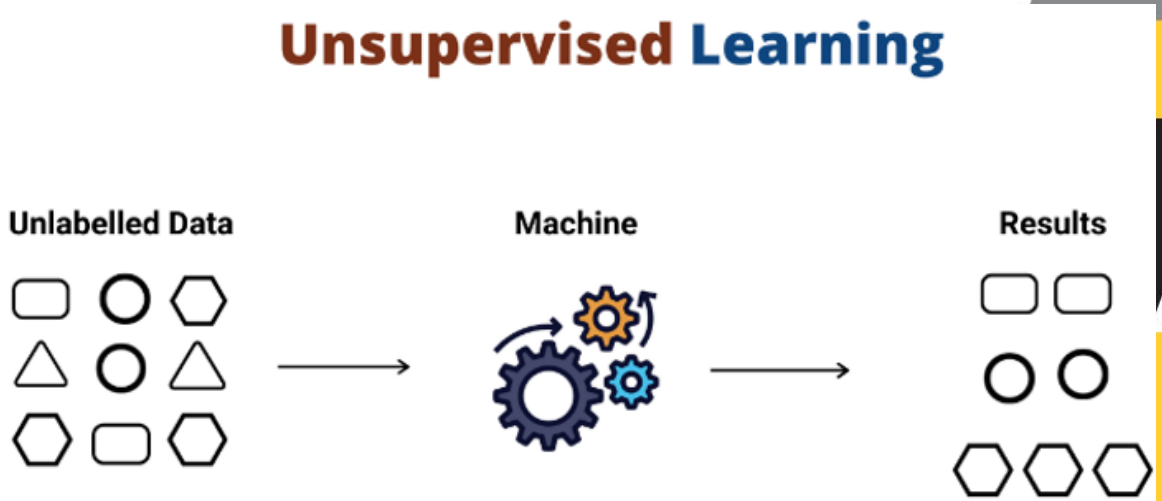
RL

- First, we need to know the difference between supervised and unsupervised learning:
  - Supervised learning: learn from examples
    - Text examples: is this message important?
    - Image classification: is this image a cat or a dog or none?
    - Regression problems: given the previous information (or from sensor reading), what will be the weather tomorrow?
    - Sentiment analysis: is the customer satisfied? Besides a yes or a no, how to understand true customer satisfaction?
    - How do we tell a function to map an input and generate an output based on examples? And then, how can we create a model?



RL

- Unsupervised learning:
  - There is no previous information (no examples)
  - One example: data is put into clusters to find patterns
- In supervised learning, we will use:
  - Labels, also referred as "**training data**"
- However, in RL we will use a combination of both:
  - "some" initial examples, and
  - Unknown predefined labels (no training data)



RL

- Two things to keep in mind:
  - When implementing RL:
    - Solutions are “ok”, so there is no need to keep improving, is this correct?
      - Our function (input-processing-output: model) is suffering from correlation
    - Solutions are “excellent”, however, metrics show the opposite
      - Correlation exists, and
      - Our function (input-processing-output: model) is being greedy

Elements of RL

- Agent:
  - Is in charge of interacting with the environment by executing actions, observing the results of these actions, and receiving rewards for those actions.
- Environment:
  - Everything outside of the agent (where the agent lives, the universe).
- Rewards:
  - Scalar value obtained that tells the agent how well its behaving.
    - Can be negative or positive
    - Large or small

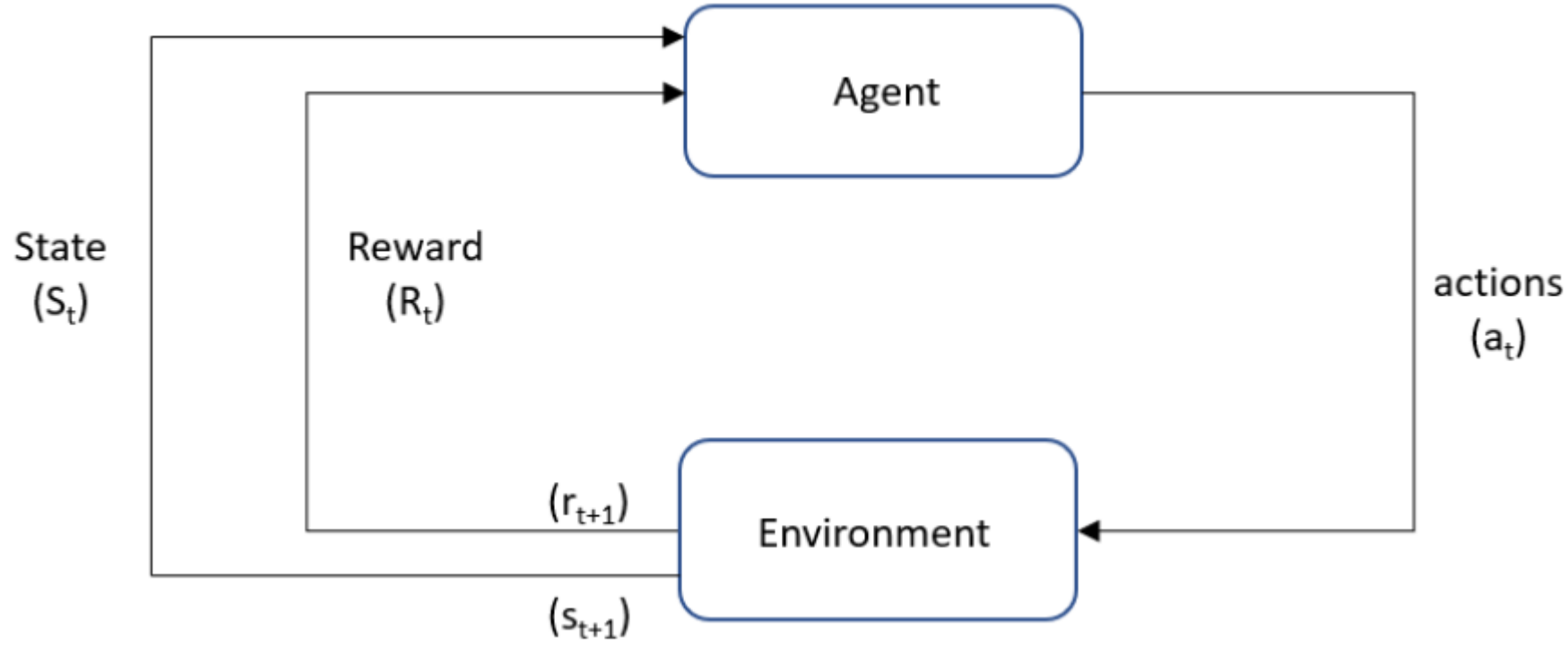
Elements of RL

- Rewards:
  - Are received periodically
    - For our purposes, we will reward the agent once one cycle is complete
  - Main goal:
    - To achieve the largest accumulated reward over its sequence of actions

- Actions:
  - Things that the agent can do in the environment
  - Actions can be:
    - Discrete:
      - Such as: move right
    - Continuous:
      - Such as: steer right (angle and direction) and move forward (slow, fast?)

Elements of RL

- Observations:
  - Information that the environment provides to the agent. The agent can observe the result of an action:
    - Two channels of information for the agent:
      - Rewards
        - Measures the result of the action: good, bad
      - Observations
        - Sees any change in the environment after executing an action



- State: What it's observed
- State space: All possible states for a system
- Action space: set of actions. Has to be finite.

- Policy:
  - Set of rules that controls the agent's behavior
  - In the case of the robot:
    - Move forward, no matter what
    - Go around obstacles by checking if the previous forward action failed
    - Spin, no matter what

    - Formally, a policy is:

      - $\pi(a|s) = P[A_t = a|S_t = s]$ (probability distribution over actions for every possible state
      - Is not concrete, use probability to introduce randomness

Q-values

- Defined for state – action
- Tied to rewards
- Based on Bellman Equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q\,(S_{t+1}, a) \right]$$

  - α: constant of step parameter (weighted average of rewards that had been given to the agent
  - γ: constant for max Q-value

Q-learning vs. SARSA

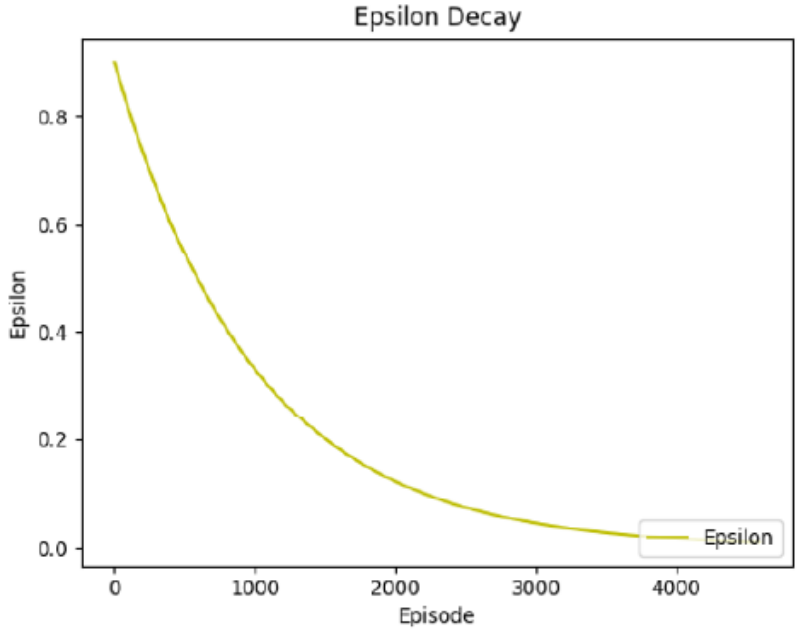
- In Q-learning:
  - Learning is done through prediction of future values

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q\,(S_{t+1}, a)\right]$$

Approximates next values.
As a preventive measure: γ (0 – 0.99…] defines how much of the predictions are implemented. Also known as “discount”.
(Some literature can also call the “discount” as “learning rate”. We will use the term “learning rate” for NN optimizers, not for RL.

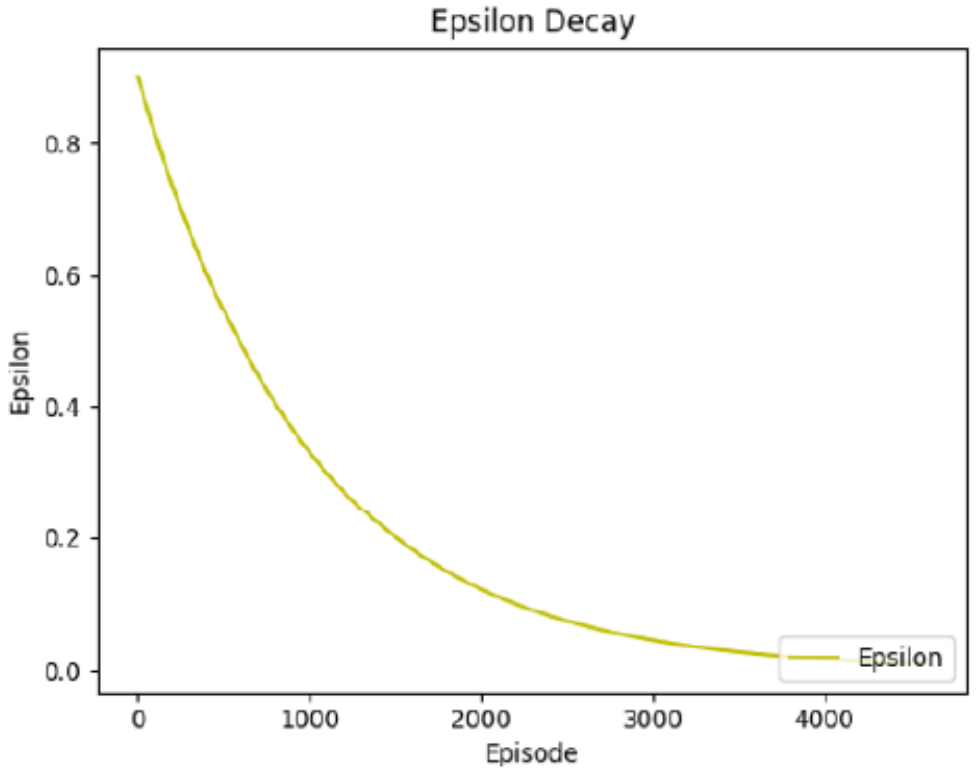
  - If using Q-learning:
    - Other hyperparameters need to be modified, besides gamma and alpha:
      - Number of episodes
      - Epsilon
      - Epsilon decay

- Epsilon decay:
  - Combination of number of episodes and epsilon
  - Tells the agent to:
    - Explore the environment, or
    - Exploit the environment
  - Explore the environment:
    - The agent will execute more actions:
      - Pros:
      - More knowledge of the environment learned by the agent
      - Cons:
      - Takes time, computationally intensive
  - Exploit the environment:
    - The agent will execute some actions:
      - Pros:
      - Faster convergence
      - Cons:
      - Agent is not efficient: knowledge of the environment is not fully learned



Ideal representation of exploration/exploitation

- To achieve a balanced scenario:
  - Hyperparameter tunning:
    - Vary the number of episodes
    - Vary the value of:
      - Epsilon: 0.1 exploits, 0.9 explores
      - Epsilon decay: How fast/slow the decay is set (or, how big the step is for decay).
        - For example: a 0.99 value is a fast decay



SARSA

- State, Action, Reward, State, Action: Repeat

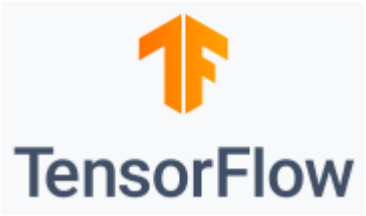
- Difference between SARSA and Q-Learning:
  - SARSA: On-policy, the policy is always the same
    - Less complex: easy to implement
  - Q-Learning: Off-policy (takes the “max” predicted value, but regulated by gamma)
    - More complex: needs to include other hyperparameters for good performance

- Example:
  - Define an environment that will give the agent random rewards for a limited number of steps, regardless of the action executed by the agent.

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

TensorFlow and TensorBoard

TensorFlow



- Open-source library for AI
  - Widely implemented for building and training neural network models.
  - We can specify:
    - Convolutional layers
      - 1, 2, or 3-dimensional convolutional layers
    - Number of layers
    - Fully connected
    - Some hyperparameters:
      - Dropout
      - Number of neurons
      - Optimizer parameters

TensorFlow

- Two important methods:
  - model.compile: Configure/build the learning process
    - Specify:
      - Optimizer
      - Loss
      - Metrics: ‘accuracy’
        - Accuracy: performance of the model
    - A typical model.compile method would take as parameters:

```
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy', metrics = ['accuracy’])
```

  - Adam (other optimizers: SGD, AdaGrad, RMSProp, and others):
    - Method to optimize the weights on each neuron in the network
  - Sparse_categorical_crossentropy (others: MAE, MeanAbsoluteError, MSE, and others):
    - Difference between predicted and actual output; based on probability distributions. Comes from Cross Entropy (binary classification).

TensorFlow

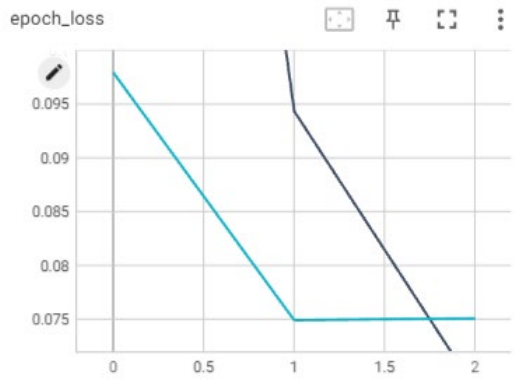
- model.fit: Trains the model and evaluates the training process
  - Specifies:
    - Training and validation data
    - Number of epochs for training
    - Callbacks: use for TensorBoard
  - A typical model.fit method would take as parameters:

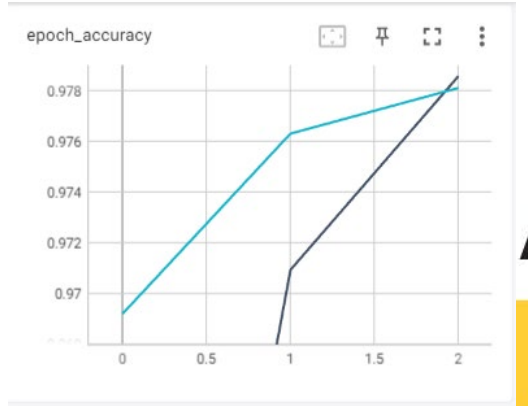
```
model.fit(x = x_train, y = y_train, epochs = 3, validation_data = (x_test, y_test),
          callbacks = [tensorboard_callback])
```

  - x, y: training data
  - epochs: number of passes for training
  - validation data: evaluate the results against these data
  - callbacks: saves TensorBoard logs to graph results of after training is complete

TensorBoard

- Illustrates the results of a training session after the model completes all epochs.
- Can show:
  - Accuracy
  - Loss
  - Customized graphs:
    - Rewards per episode per epoch in DRL
    - Penalties
  - Histograms
  - Graph (tree) representation of the model



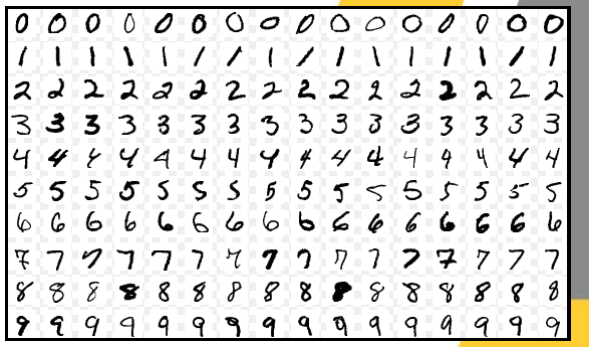


TensorBoard

- To save logs for TensorBoard:
  - Use the default current working directory
  - In the default current working directory:
    - Create a path/folder to save the logs:
    - In macOS:
      - `log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")`
    - In Windows:
      - `log_dir = "logs\\fit\\" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")`

- To initialize a TensorBoard session:
  - In Visual Studio Code:
    - In macOS: **command + shift + P** In Windows: **ctrl + shift + P**
    - enter: Tensorboard, select -> Python: Launch TensorBoard
    - Select another folder
      - Browse and select your “fit” folder

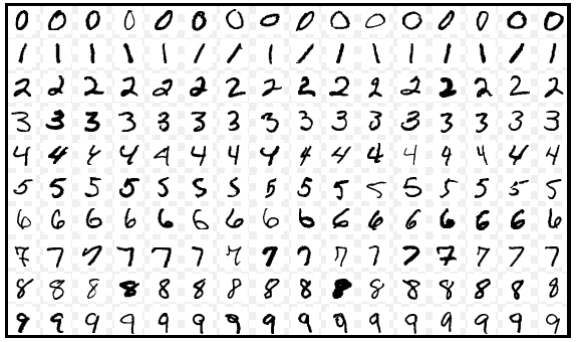
TensorFlow and CNN

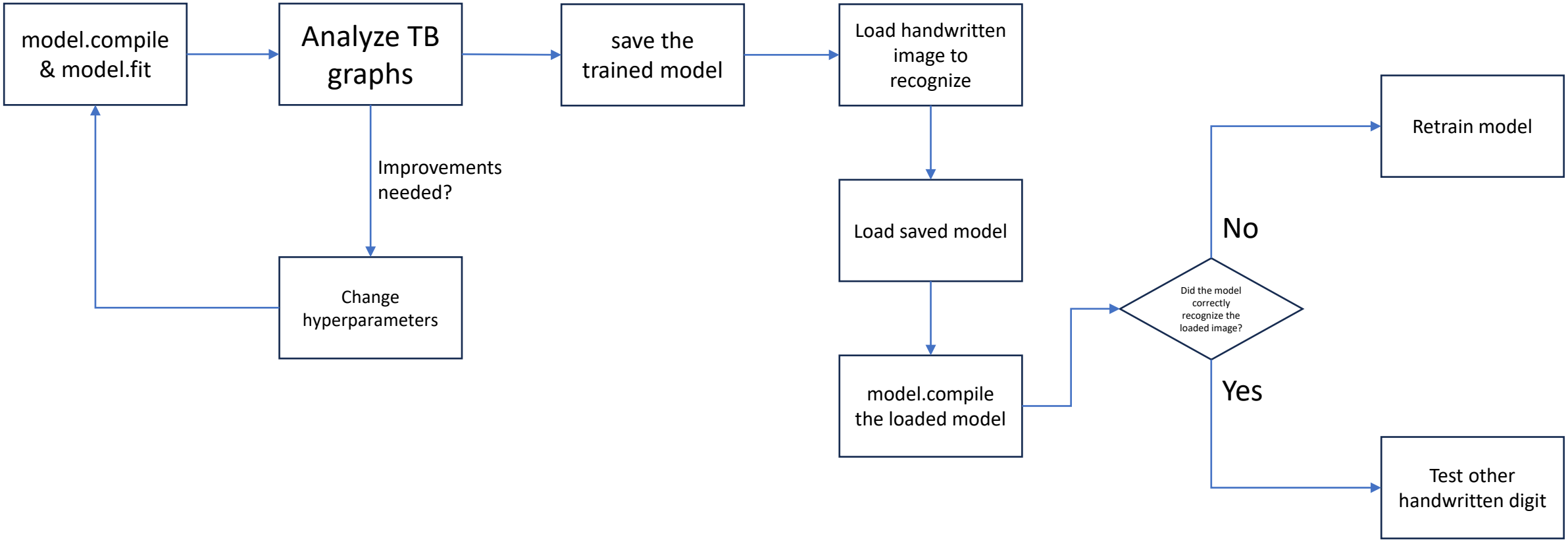


- Using the mnist database:
  - Modified national institute of standards and technology
  - Contains hand-written images used to train NN models typically used for character recognition
  - The database is made of:
    - 60,000 images used for training
      - Image and label
    - 10,000 images used for testing
      - The model should identify the image without the label

TensorFlow and CNN

- Create a model that can recognize handwritten digits:





THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Optimization

Optimization

- A network needs to have an input and an output
  - Two definitions for output:
    - Approximated output (“Predicted output” in TensorFlow syntax)
    - Labeled, target output
      - Target output calculated: model-free
      - Target output given: use labeled data
  - How close the predicted/approximated is from the labeled/target output is a **loss function,** and its value is the **loss value**
  - **Optimization**: using mathematical operations to reduce the loss value using gradients.

Optimization

- Loss functions (to calculate loss):
  - Take two parameters: Approximated/predicted output and target output
  - Some commonly used loss functions are:
    - MSELoss: (mean square error):
      - Typical for regression and approximation problems
      - Calculates differences between target and predicted/approximated outputs

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^i - \hat{y}^i\right)^2$$

    - Categorical Cross-Entropy: Typical for classification/categories problems

$$CEL = -\frac{1}{n}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} \cdot logp_{ij} \; or \; -\frac{1}{n}\sum_{i=1}^{N} log_{P_{model}}\left[y_i \in C_{yi}\right]$$

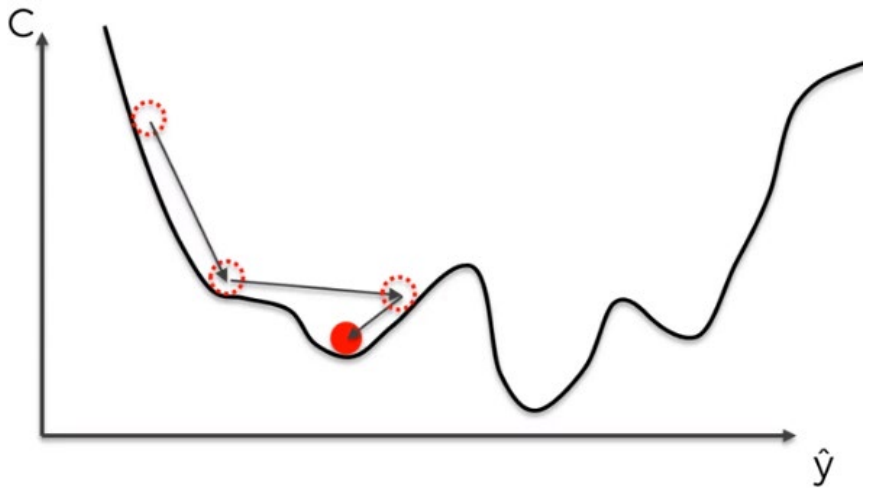
Optimization

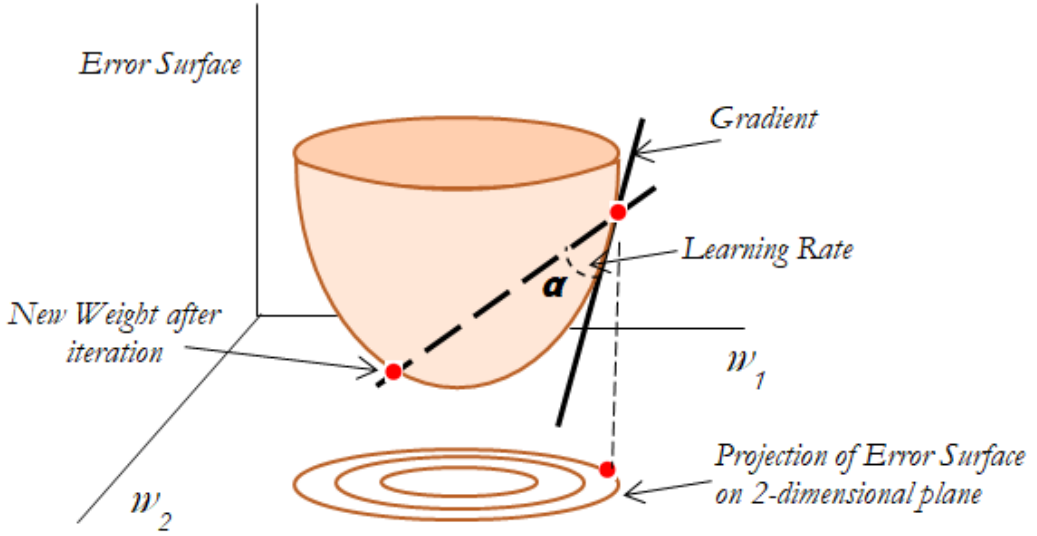
- Now that we have a loss value, we need to optimize to improve the model in training.
  - Gradients: vector that denotes the fastest increase and direction

- The optimization process:
  - Take the gradients of model parameters
    - Weights and biases
  - Change these parameters in order to decrease the loss value
  - Once the parameters are recalculated:
    - Update the parameters through **backpropagation**

Optimization

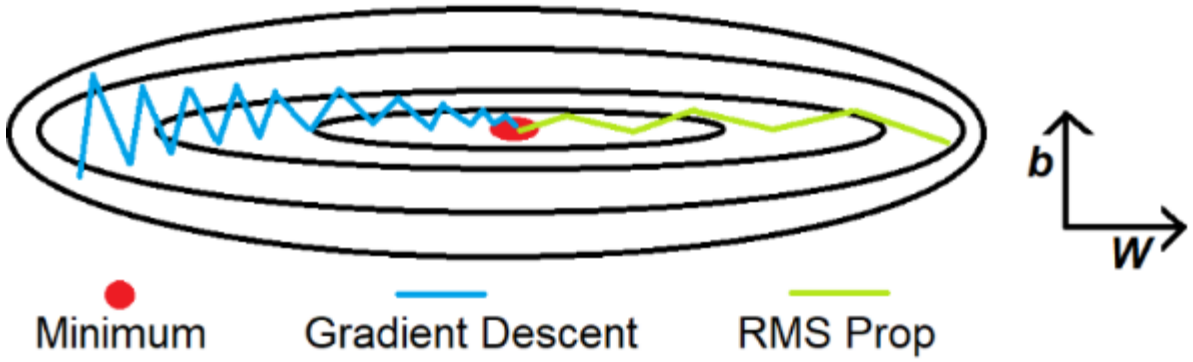
- Typical optimization methods:
  - SGD: stochastic gradient descent
    - Stochastic: randomly choose from the batch
    - **Variable parameter: learning rate**





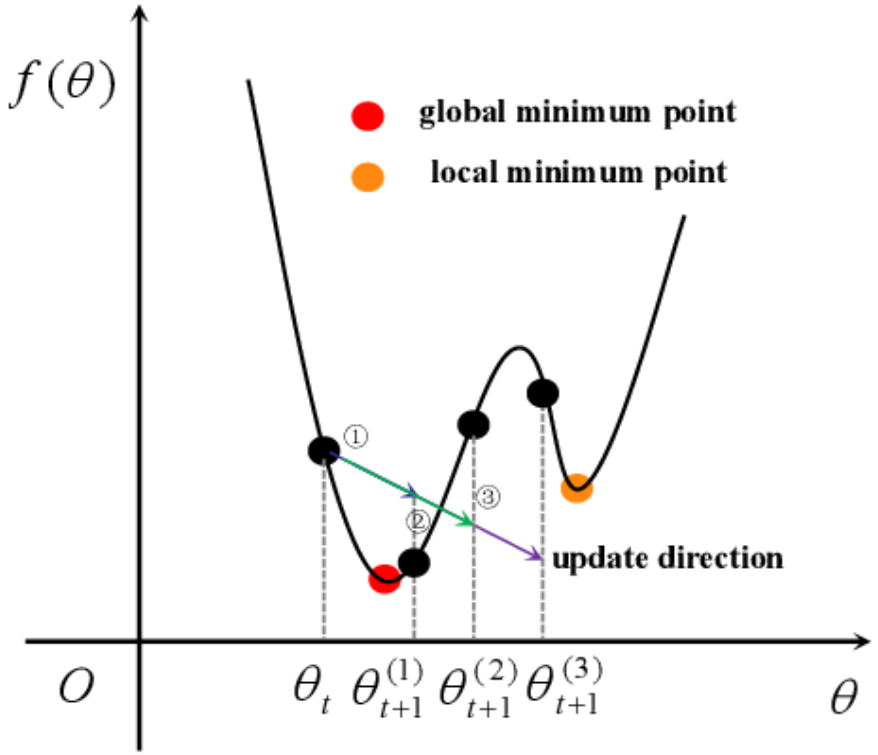
Optimization

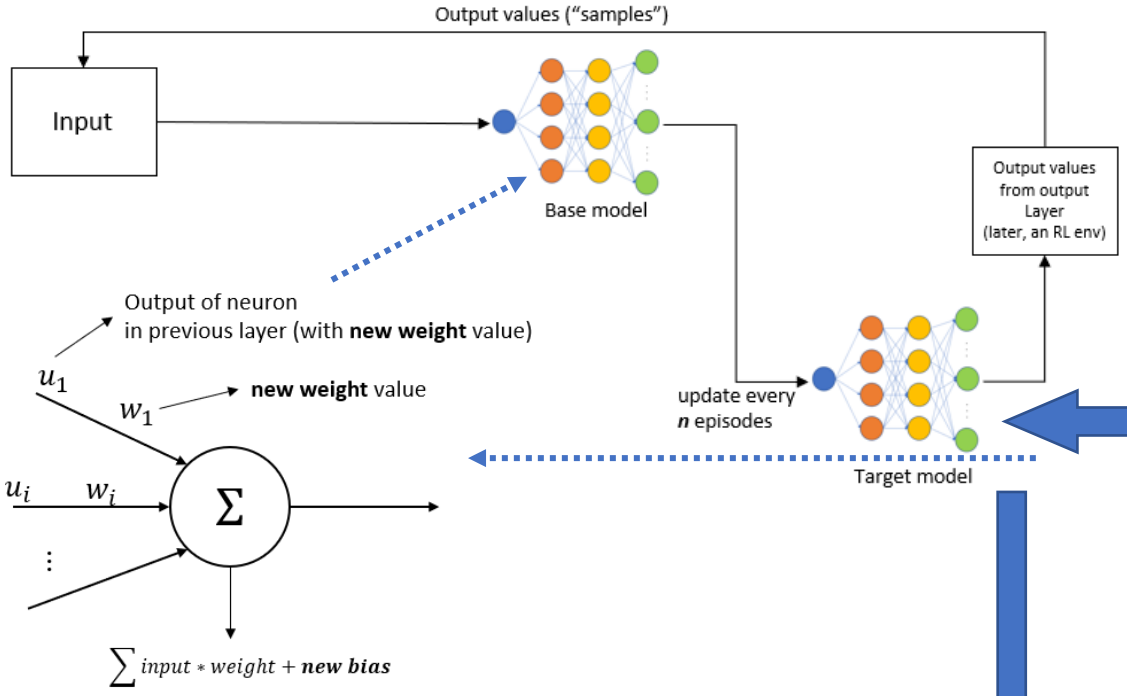
- RMSProp (Root mean squared propagation):
  - Based on AdaGrad (adaptive gradient)
    - Calculation of the step size (learning rate, LR) is “automatic”. Adapts the LR according to each parameter
    - Can slow down the process
  - RMSProp: uses a decay for learning rate using moving average (or weighted average)



Optimization

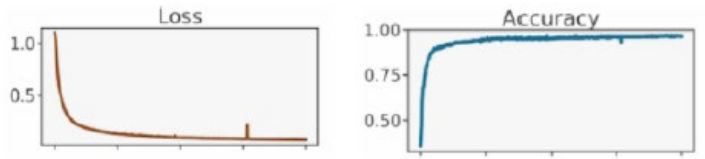
- Adam (adaptive moment)
  - Combination of AdaGrad and RMSProp
  - The learning rate is adapted in two calculations (moments):
    - Uses the average of the second moment (through the exponential moving average of gradients and square gradients)
    - Controlled decay: beta 1 & beta 2





backpropagate

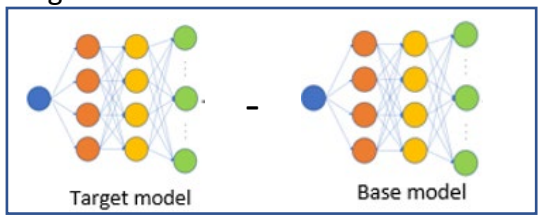
Loss (and accuracy) improved.
Update parameters now!



Using an optimization method

Using a loss function

Loss value =



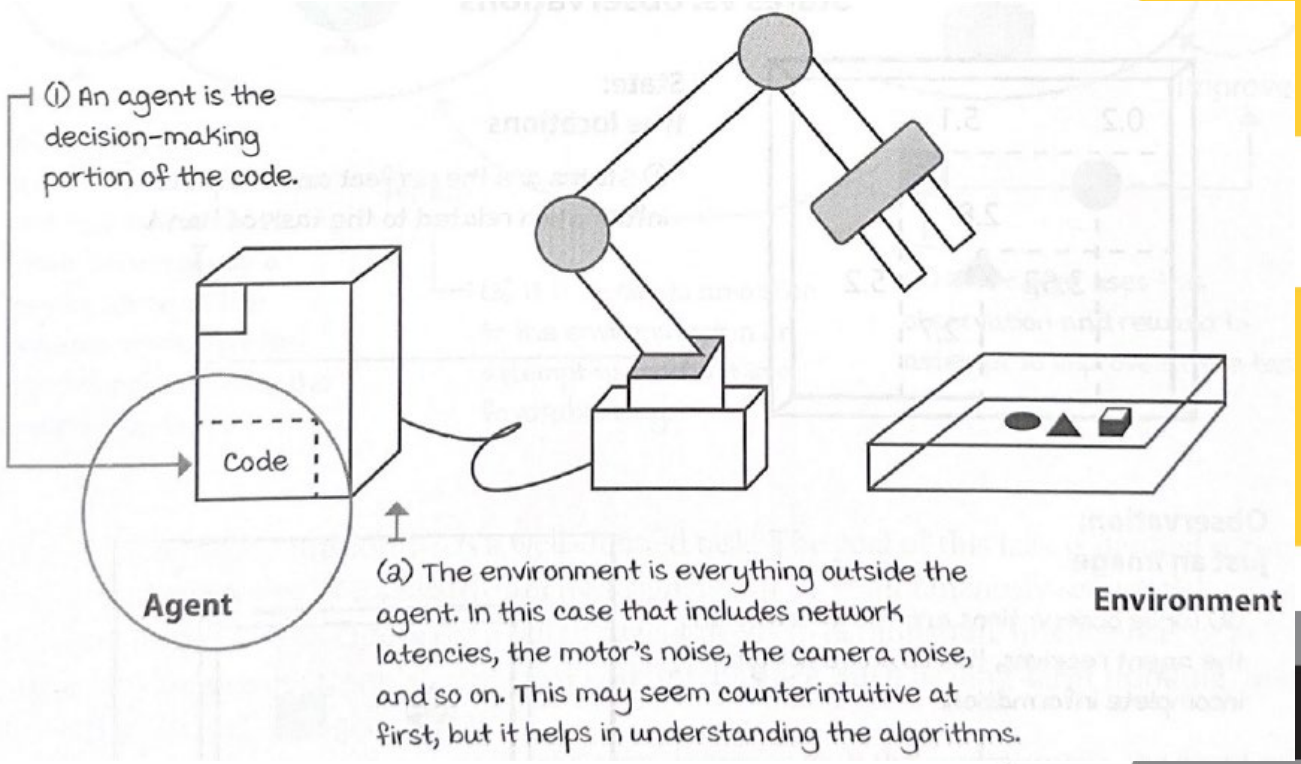
Loss values are bad: Optimize!

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Introduction to Neural Networks

In RL

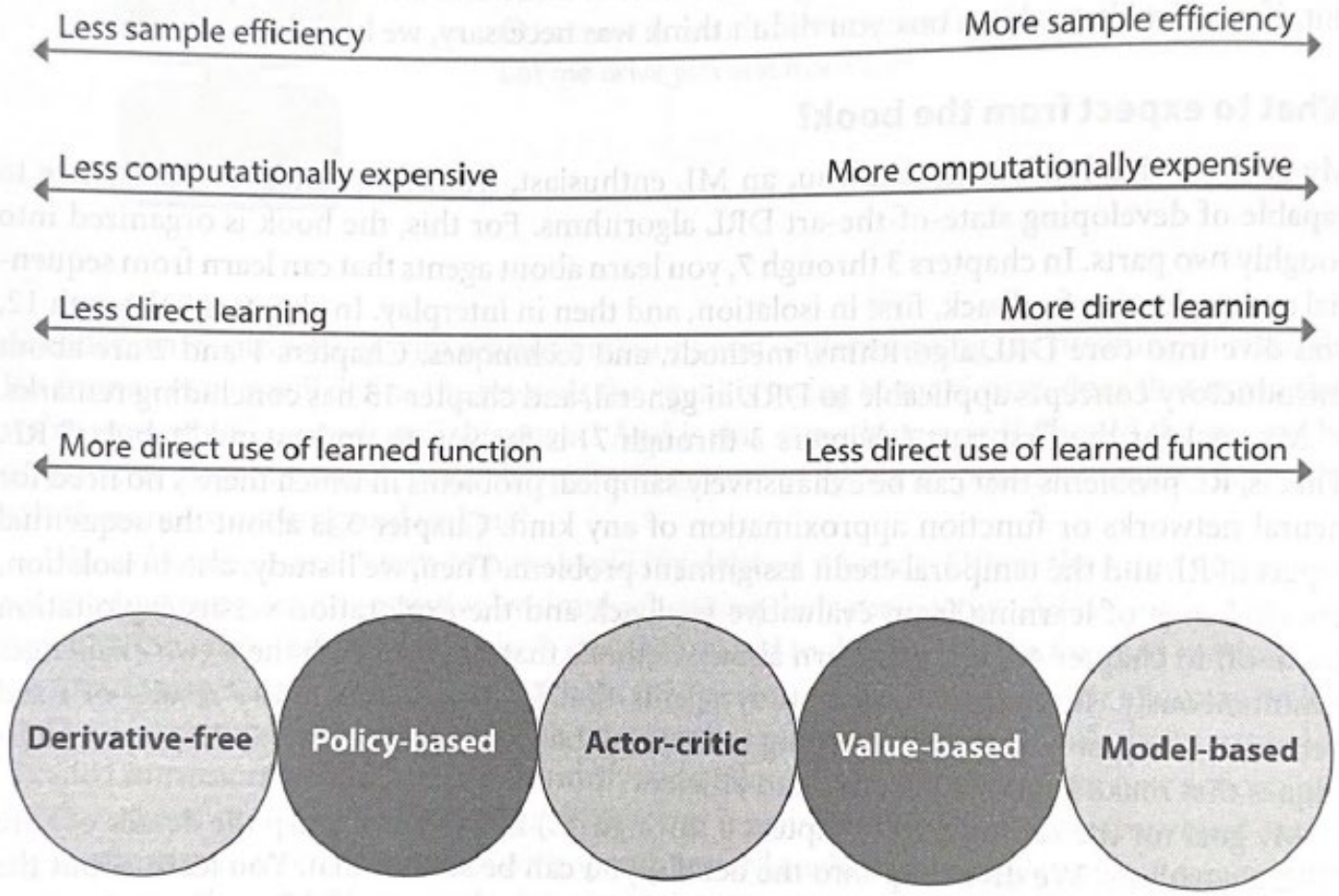
- Three approaches to learning:
  - Policies: map observations to actions
    - Depending on configuration:
      - Off-policy: Q-Learning
      - On-policy: SARSA
  - Models: learn the environment based on a model
    - Target model: goal of the agent
    - Base model: interaction of the agent with the environment (computationally intensive)
  - Value functions: learn to estimate the reward-to-go on mappings



- As more observations, more Q-values, more new states are generated:
  - Interaction of the RL agent and environment grows significantly.
  - An RL approach for more complex environments will be computationally expensive.

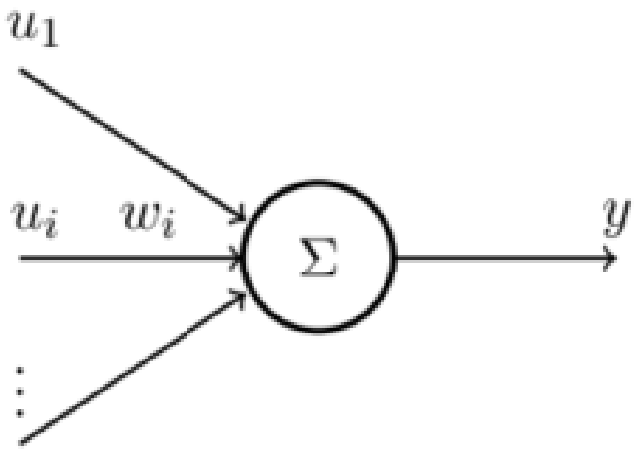
- Since RL is based on “trial – error”, an RL agent will have difficulties in more complex environments
  - The “prediction” section of Q-learning
    - Also based on “trial – error” by applying (state, action) from previous observations
- To solve this, implement ***approximations***:
  - Neurons

- More resources are needed depending on approach:



Neuron

- A neuron consists of:
  - An input, **u**
  - An output, **y**
  - An activation signal
  - A threshold, **θ**
  - A weight**, w**
  - *bias, **b**, can be part of the neuron, it can also be considered a hyperparameter



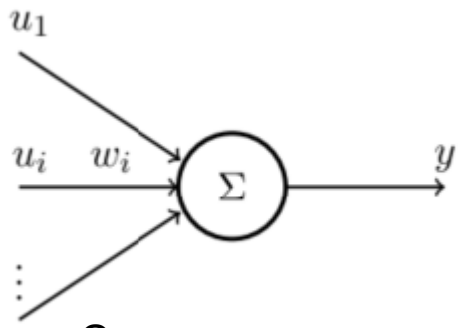
- Mathematically:

$$y = f(x), x = \sum_{i=1}^{n} w_i u_i$$

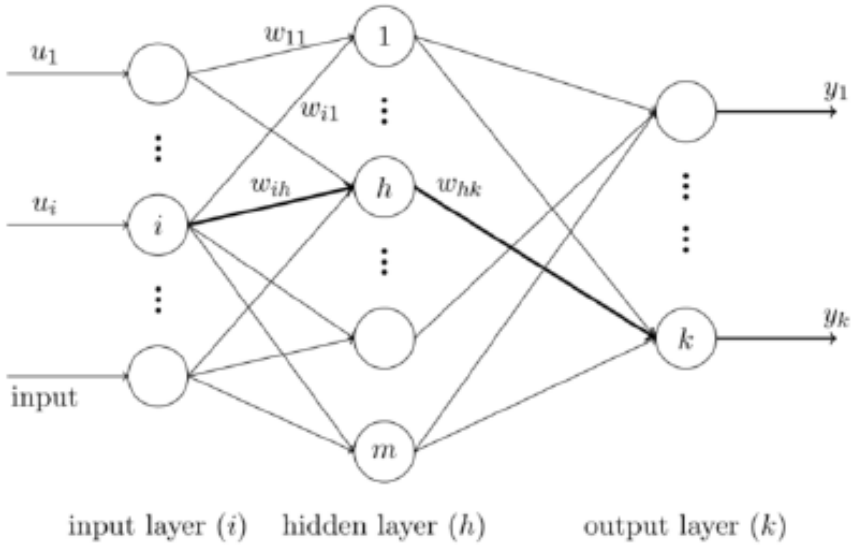
Where: x is a weighted sum, and f(x) an activation function depending on the parameter θ

Neuron

- When several neurons are configured in layers, and all neurons are connected to each other, it's known as a:
  - Fully connected neural network
    - In the simplest form, this is called: **Artificial Neural Network, ANN**
    - A model using an ANN with multiple (usually more than 2 or 3) NN layers is referred to as a model using *Deep Learning*.



One neuron



Neural network

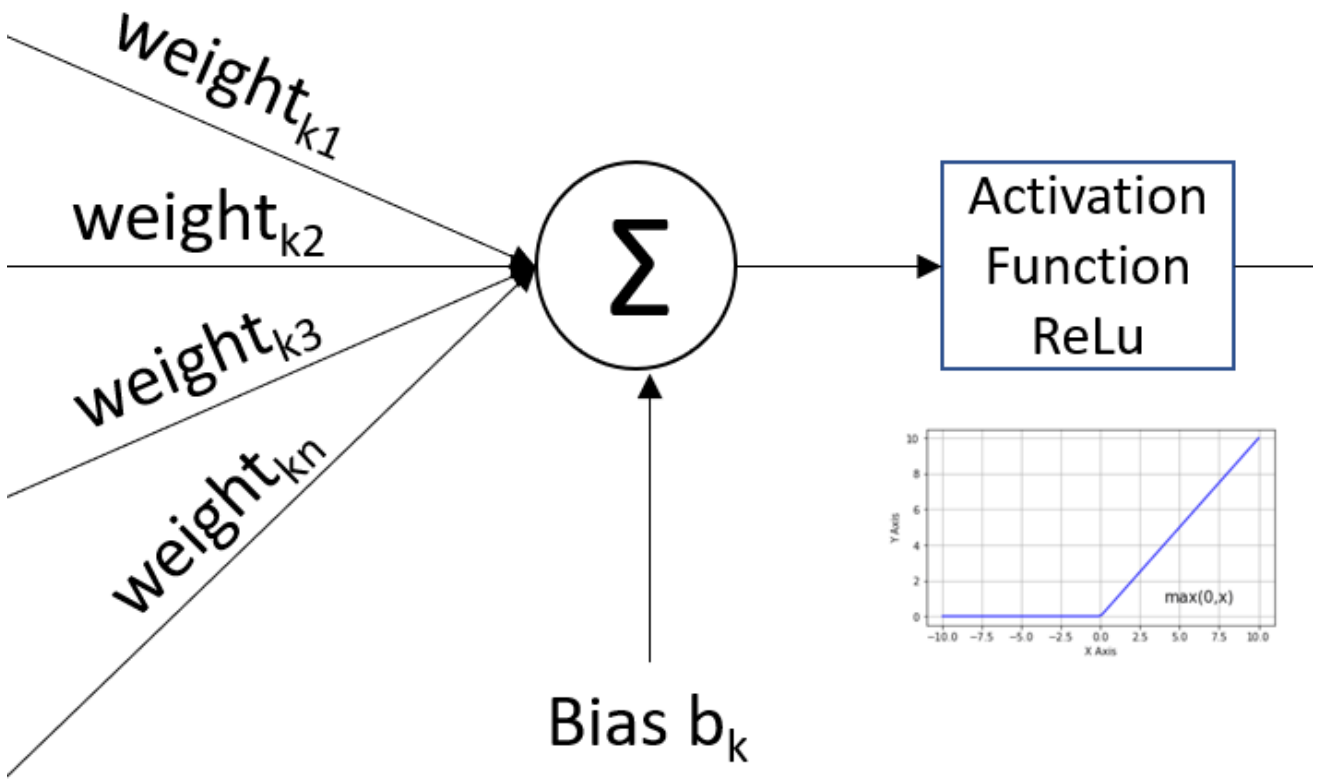
Layers

- In an ANN, in its basic structure: three layers
  - Input Layer:
    - Data coming from “the environment”
      - In DRL these will be observations from the agent
      - In Deep Learning:
        - Labeled data, i.e.
    - Depending on application:
      - Convolutional (& pooling layer): format the input data
        - Convolutional Neural Network, CNN
      - Recurrent: memory-based implementation
        - Recurrent Neural Network, RNN
  - Hidden Layer(s)
    - Where all approximations occur
  - Output Layer
    - Results of the approximations

Approximation

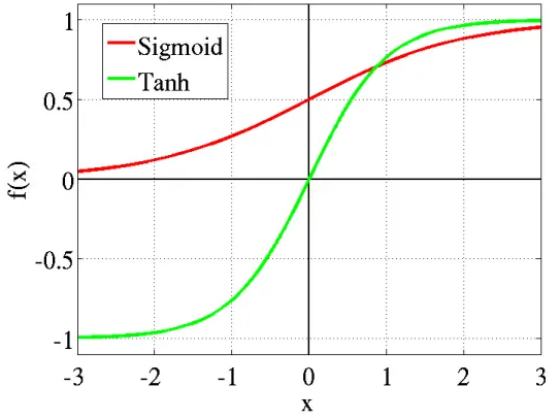
- In one neuron, calculations are occurring:
- Neurons in the hidden layer:
  - Input: weighted sum of neurons in previous layers
  - Weighted sum: input * weight + bias (bias can be optional, however, its is most likely to be included in the calculation)
  - Activation: check if the final weighted sum is within threshold parameters
    - If final weighted sum is within parameters:
      - Neuron remains active
    - If not:
      - Send weighted sum to next layer
      - Deactivate neuron
        - Neuron is no longer used on the next episode (epoch)
- Output layer:
  - Results of approximations:
    - transformed into formatted data

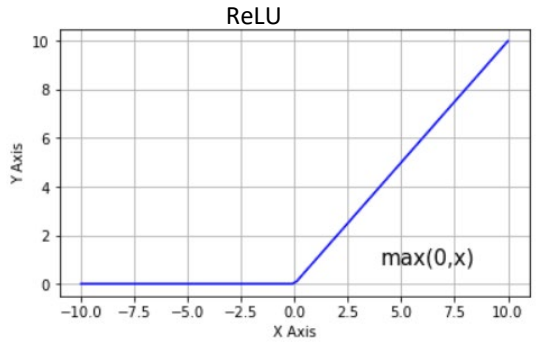
Hidden Layer



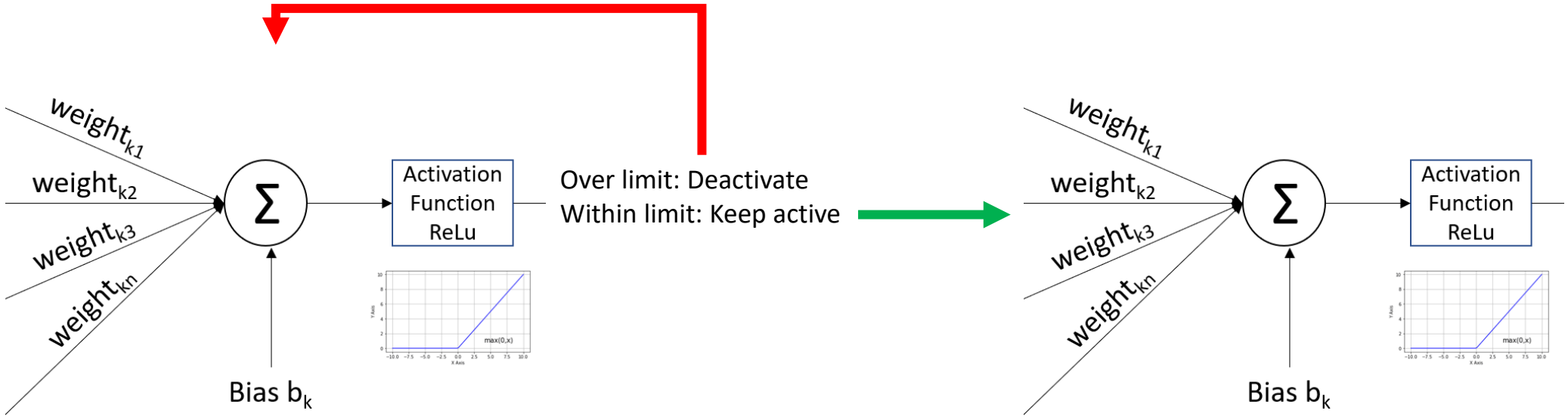
Activation

- Activation is used to decide whether the neuron in question should be kept active or not:
  - Methods:
    - Softmax: based on probabilities
      - Calculates a probability
    - Sigmoid: when the output needs to
      - based on probabilities
    - Tanh: similar to sigmoid, except that
      - Tanh will accept negatives
    - Rectified Linear Unit (ReLU):
      - f(x) is 0 when x is 0



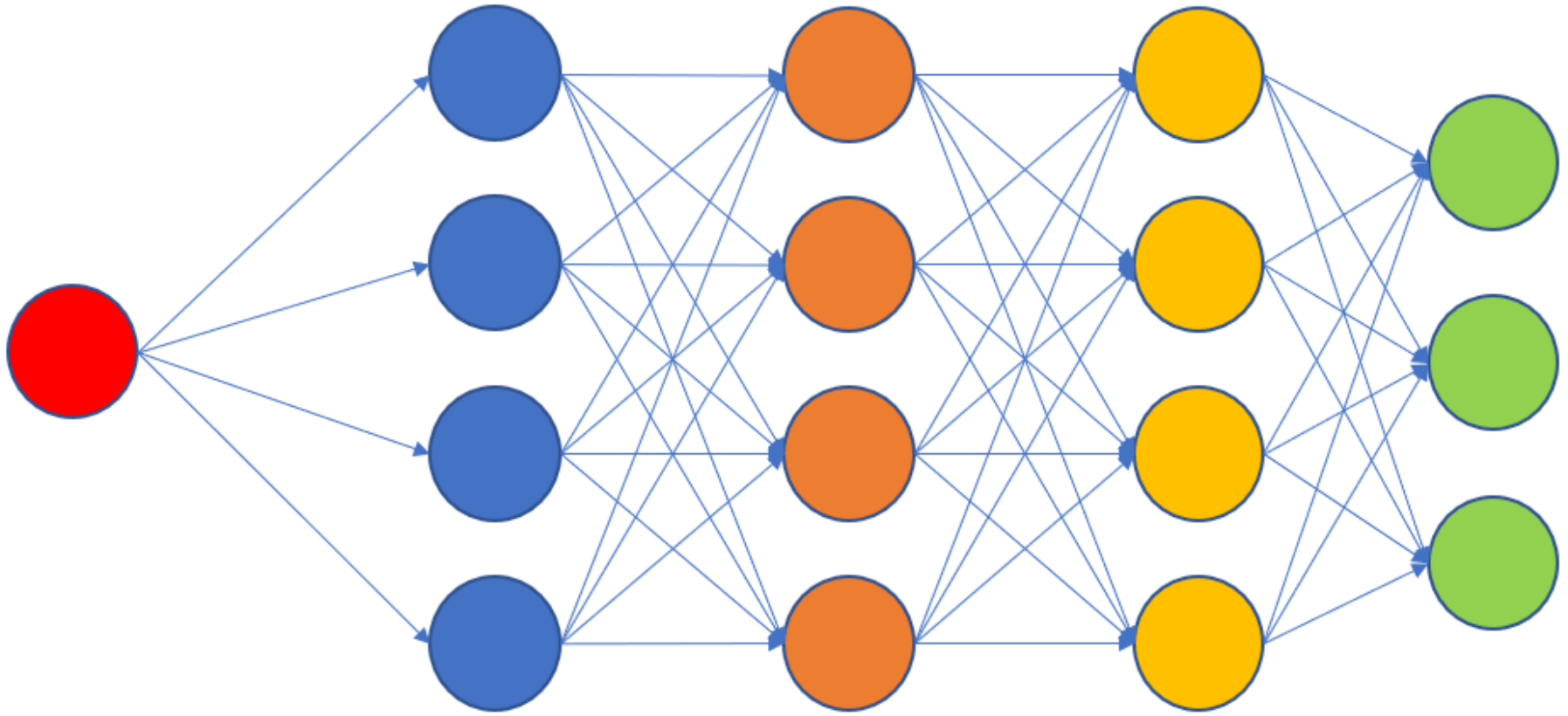


Activation


weight$_{k1}$
weight$_{k2}$
weight$_{k3}$
weight$_{kn}$
Σ
Activation Function ReLu
max(0,x)
Bias b$_k$
Over limit: Deactivate
Within limit: Keep active
weight$_{k1}$
weight$_{k2}$
weight$_{k3}$
weight$_{kn}$
Σ
Activation Function ReLu
max(0,x)
Bias b$_k$

- Before approximations begin:
  - Data needs to be “transformed” into proper neural input
    - Convolutional layer (& pooling layer):
      - Filtering
      - Pooling
      - Flatten & dense

Representation of a fully connected ANN



In CNN

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Autoregressive Models

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI

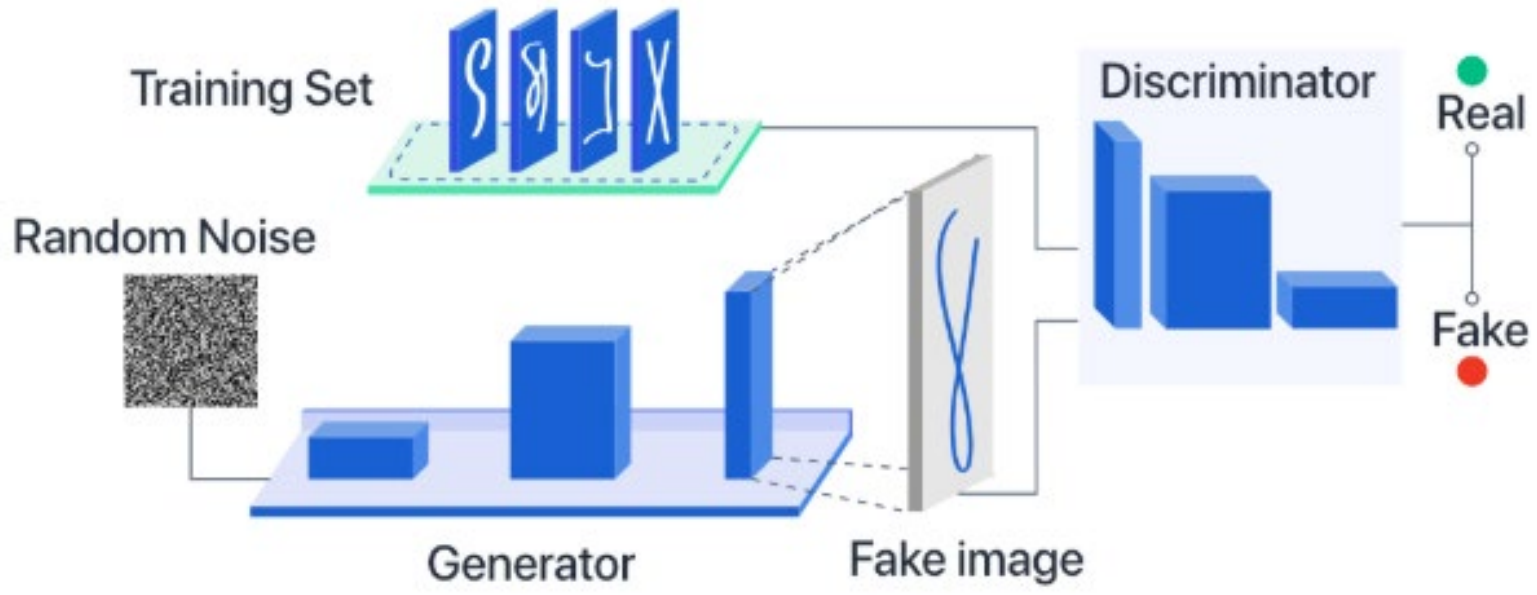
Autoencoders and GANs

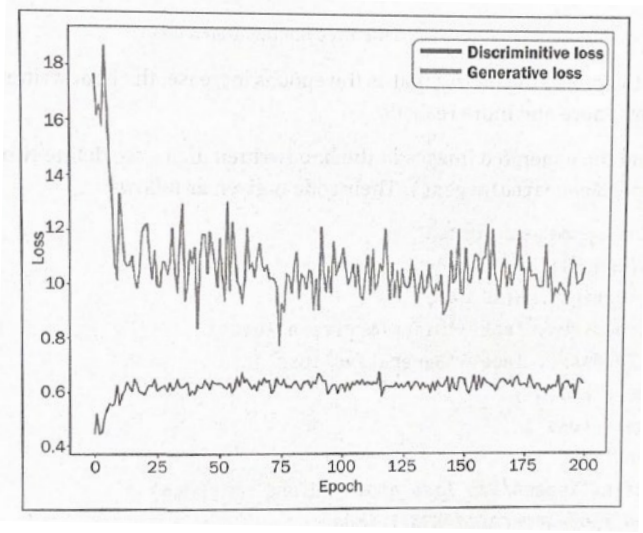
- So far, we have studied a family of generative models:
  - Autoencoder:
    - Samples from latent space
    - Learns how to decode back into the original domain
    - More advanced model: Variational Autoencoder (VAE)

  - Generative Adversarial Model (GAN):
    - Two adversaries:
      - Generator & Discriminator
      - Generator trues to convert random noise into observations that look as if they have been sampled from the original dataset (this is forgery)
        - Similar to the Decoder in AE, samples from latent space using multivariate standard normal distribution
      - Discriminator predicts if an image is real or fake

GAN

- In GAN models:
  - Two networks:
    - Discriminator network: a standard CNN
      - Tries to classify the image as real or generated
    - Generator network:
      - gets its updates from backpropagated values from the discriminator





At first, we would want the D_Loss to be high and G_Loss to be low.
However, a “good” trained model should have low D_Loss and low G_Loss. This can be a sign of the Generator being able to “trick” the Discriminator.

Autoregressive Models

- Family of generative models that treats a problem as a sequential process.
- They condition predictions based on previous values in sequence rather than on latent space.
- In conclusion:
  - They attempt to **model** the data-generating distribution rather than an approximation of it (as in autoencoders)
  - Some models:
    - LSTM: Long short-term memory: Good for text
    - PixelCNN: Good for image
    - Transformers: Good for text

Autoregressive Models: LSTM

- LSTM is a RNN (recurrent neural network):
  - RNNs contain a recurrent layer (or cell) that can handle sequential data
    - an output of the layer at a particular timestep be part of the input in the next timestep
- Since LSTM is designed to work with sequential data, it fits well with text data.
- Key differences between working with text and image data:
  - Text data is composed of discrete chunks
    - We cannot change the word *cat* to more *cat*
  - Image data is composed of pixel values
    - Pixel data gets filtered, we can backpropagate, calculate loss, etc., to make a pixel more blue.

Autoregressive Models: LSTM

- Text data has a time dimension and no space dimension
  - Order of words is highly important
- Image data has two spatial dimensions but no time dimension
  - Images can be flipped without changing the content
- Text is highly sensitive to small changes in individual units
  - Changing a few words can change the context
  - Changing a few pixels in an image can still be recognizable

Autoencoders: LSTM – Tokenization

- Tokenization:
  - Process of splitting the text into individual units, i.e., words or characters

- When using word tokens:
  - Lower & upper case matters
  - Rare words can be replaced with a token for unknown word to reduce the number of weights in the NN.
  - Words are reduced to its simplest form
  - Tokenize punctuation or eliminate punctuation at all

Autoencoder: LSTM – Tokenization

- When using character token:
  - Generate new sequence of characters to form new words outside of the training vocabulary
  - The vocabulary is smaller – fewer number of weights to be learned by the NN

- Tokenization process:
  - For example: lowercase tokenization, without word stemming, punctuation is tokenized also:

```
def pad_punctuation(s):
    s = re.sub(f"([{string.punctuation}])", r" \1 ", s)
    s = re.sub(" +", " ", s)
    return s

text_data = [pad_punctuation(x) for x in filtered_data]

text_ds = (
    tf.data.Dataset.from_tensor_slices(text_data).batch(BATCH_SIZE).shuffle(1000))
```

1. Pad punctuation marks to treat them as separate words

2. Convert to TensorFlow dataset

3. Create a Keras TextVectorization layer to convert text to lowercase
Give the most 10,000 prevalent words a corresponding integer token
Trim or pad the sequence to 201 tokens

```
vectorize_layer = layers.TextVectorization(standardize="lower",
    max_tokens=10000,
    output_mode="int",
    output_sequence_length=200 + 1,)
```

4. Apply the TextVectorization layer to the training data

```
vectorize_layer.adapt(text_ds)
```

5. Store a list of the word tokens

```
vocab = vectorize_layer.get_vocabulary()
```

Autoencoder: LSTM – Tokenization

- For example:
- “Recipe for Ham Persillade with Mustard Potato Salad”
- Tokenized:
  - [26 16 557 1 8 298 335 189 …
  - TextVectorization Layer: creates a “map” (sort of an index, based on frequency)
    - 0:
    - 1: [UNK]
    - 2: .
    - 3: ,
    - 4: and
    - 5: to
    - 6: in
    - 7: the
    - 8: with
    - 9: a
    - …

Autoregressive Models: LSTM – Predict next sequence

- We need to train the LSTM model to predict the next word in a sequence, given a sequence of words preceding
- For example:
  - Feed model the tokens for: “grilled chicken with boiled ____”
  - Expected output: potatoes
  - Rather than: bananas

- One way to do it:
  - Shifting the entire sequence by one token to create our own target variable

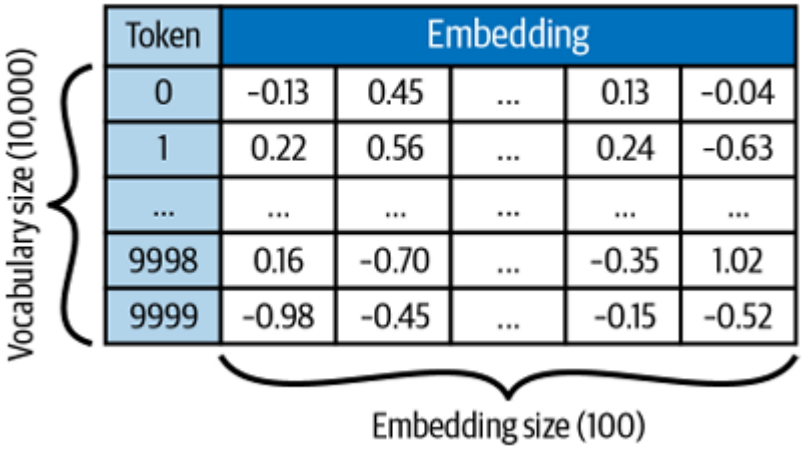
Autoregressive Models: LSTM – Predict next sequence

- The LSTM architecture:
  - Input of the model: sequence of integer tokens
  - Output of the model: probability of each word in the 10,000-word vocabulary appearing next in the sequence.

  - We need two layers:
    - Embedding
    - LSTM

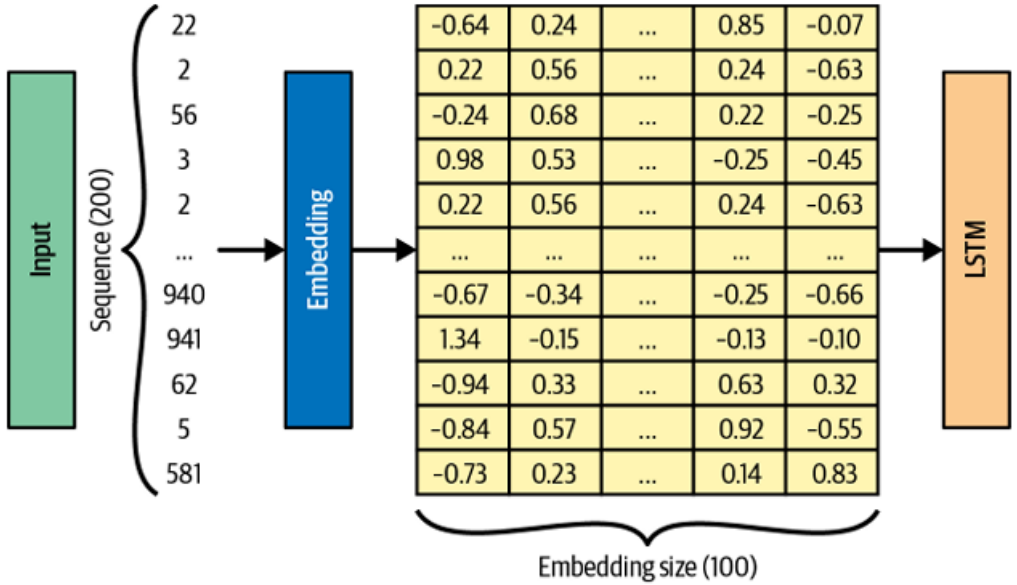
Autoregressive Models: LSTM – Predict next sequence

- Embedding layer:
  - In an essence, it is a look-up table that converts each integer token into a vector of length “embedding size”



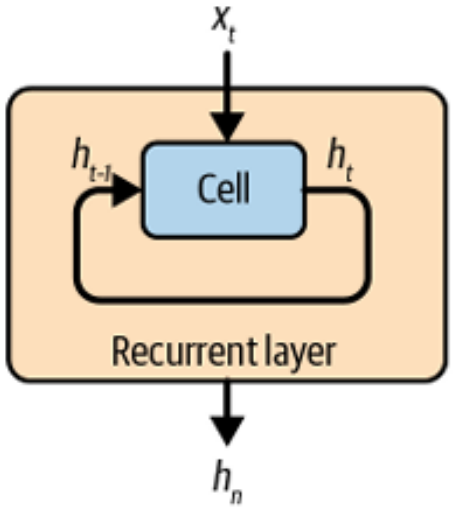
Autoregressive Models: LSTM – Predict next sequence

- The input layer passes a tensor of integer sequences to the embedding layer
- The embedding layer outputs a tensor that is passed to the LSTM layer



Autoregressive Models: LSTM – Predict next sequence

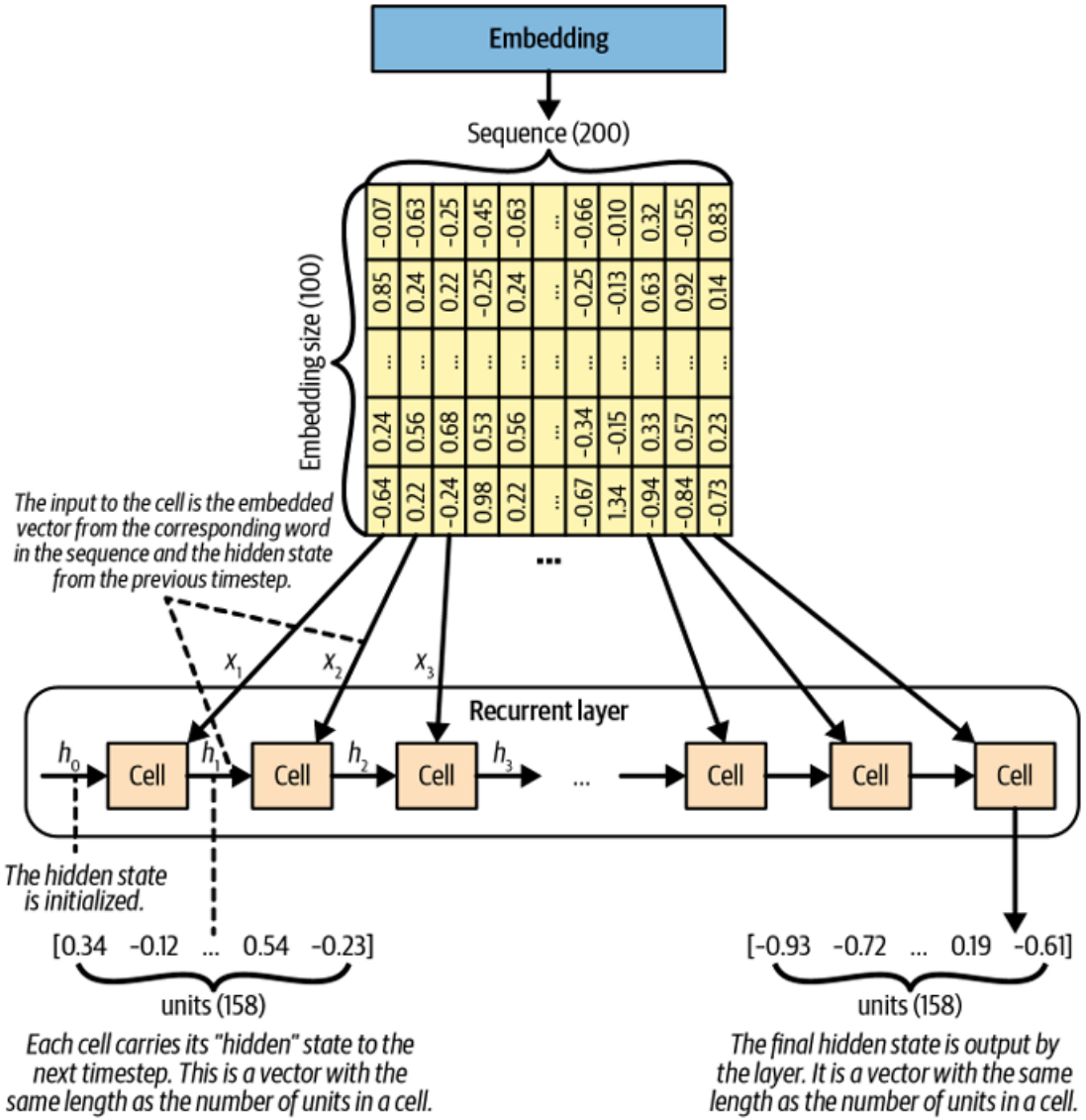
- LSTM layer
  - First, what is “recurrent”?
    - A recurrent layer consist of a cell that updates its hidden state as each element of the sequence is passed through it, one timestep at a time.
    - Hidden state:
      - A vector with length equal to the number of units in the cell
    - How does it work?
      - Given a sequential input: x_1,…, x_n
      - A hidden state: h_t
      - At timestep t:
        - Use previous value of the hidden state, h_{t-1}
        - Data of the current timestep, x_1
        - Produce an updated state vector, h_t
        - Continue until the end of the sequence
        - Output the final hidden state, h_n , to the next layer of the network



Autoregressive Models: LSTM – Predict next sequence

- Embedding and LSTM layers
  - Units and cells:
    - There is one cell defined by

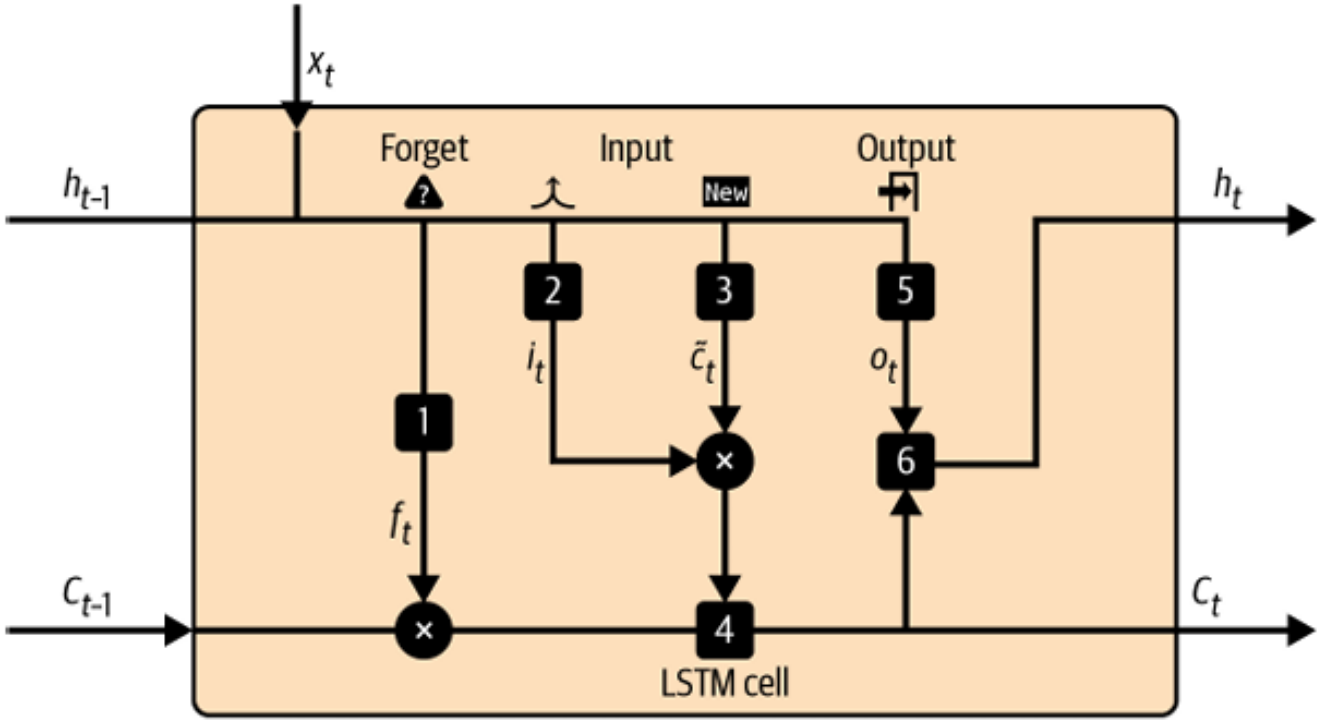
    a number of units it contains.
    (think of a prison cell that holds multiple prisoners).
    The number of units is set when defining the layer.



Autoregressive Models: LSTM – Predict next sequence

- The LSTM Cell:
  - The hidden state is updated in six steps (six neural networks):
    1. Hidden state of the previous timestep and the current word embedding x_t are concatenated and passed to the forget gate
       - The forget gate: NN with sigmoid activation
    2. Result of step 1, passed on an input gate:
       - Input gate: a NN with sigmoid activation
    3. Result of step 2, passed through a NN with tanh activation
    4. Resulting vector and cell state from step 1 are multiplied and added with the output of step 2 and step3 vector (element-wise)
    5. Result of step 4 is passed through an output gate:
       - Output gate: NN with sigmoid activation
    6. Result of step 5 is uses tanh activation to produce a new hidden state, h_t

Autoregressive Models: LSTM – Predict next sequence



1 $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

2 $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

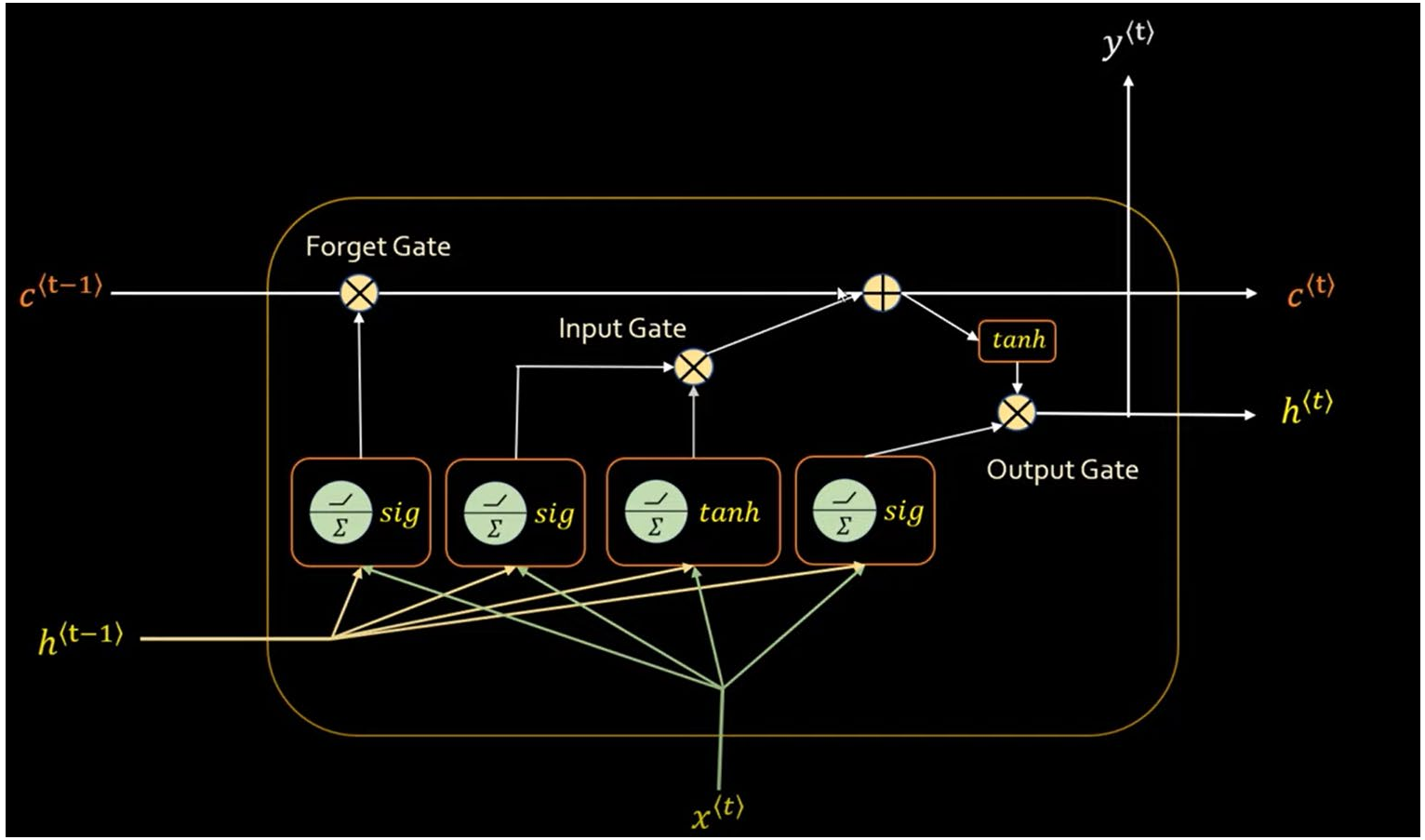
3 $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

4 $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

5 $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$

6 $h_t = o_t * \tanh(C_t)$

Autoregressive Models: LSTM – Predict next sequence



Autoregressive Models: LSTM – Training

- Most important step in training:
  - The Dense layer transforms the hidden states at each timesteps into a vector of probabilities for the next token
  - The overall Model predicts the next token, given an input sequence of tokens. It does this for each token in sequence.

Autoregressive Models: LSTM – Training

- Notice the sequence of layers:
  - 1. input
  - 2. embedding
  - 3. LSTM
  - 4. Dense

```
inputs = layers.Input(shape=(None,), dtype="int32")
x = layers.Embedding(VOCAB_SIZE, EMBEDDING_DIM)(inputs)
x = layers.LSTM(N_UNITS, return_sequences=True)(x)
outputs=layers.Dense(VOCAB_SIZE,activation="softmax")(x)
lstm = models.Model(inputs, outputs)
lstm.summary()
```

Autoregressive Models: LSTM – Predict next sequence

- Example:
  - Train an LSTM model to generate new cooking recipies

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Generative Models
Autoencoder: Encoder

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI

Generative Models

- Is a type of machine learning algorithm for data creation.
  - Data generated is similar to the data used to train a model
- Applications:
  - Image Synthesis
  - Natural language processing
  - Density estimation

- Variations of Generative Models:
  - Autoencoders
    - Variational Autoencoder
  - Generative Adversarial Networks
    - Generator vs. discriminator
  - Autoregressive models
    - LSTM (RNN)
  - Transformer
    - GPT
  - Diffusion Models
    - Works on noise generation

Image

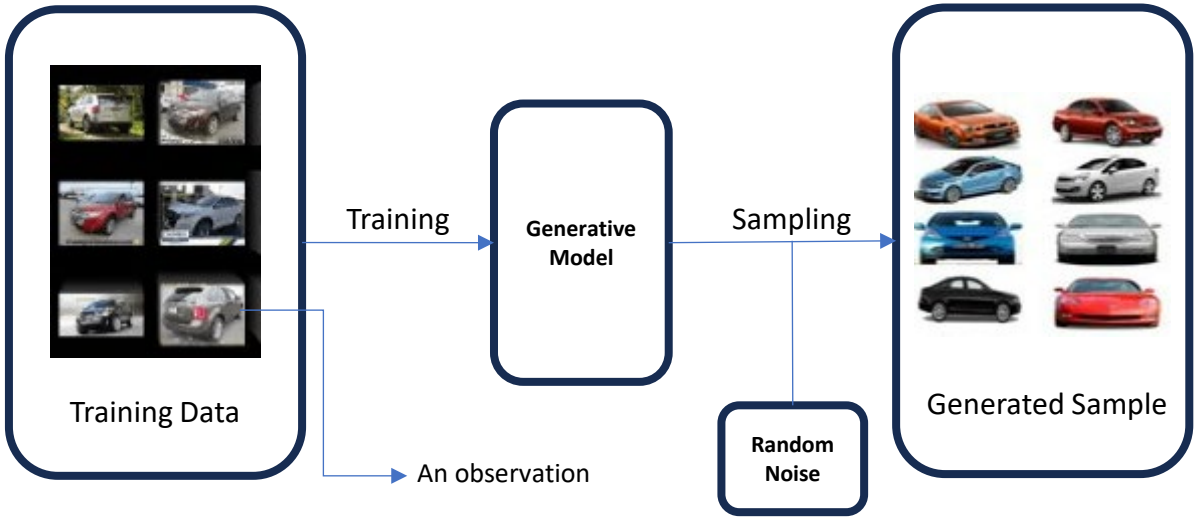
Text

Generative Models (GM)

- Train a generative model to capture rules that govern the relationship between pixels in an image of, for example, horses, cars, human faces, etc.
- Then, sample from this model to create realistic images that did not exist in the original dataset.
- Two main approaches for GM applications:
  - Image
  - Text
- Each observation consists of many features:
  - For image problems: individual pixel values
  - For text problems: individual words or groups of letters

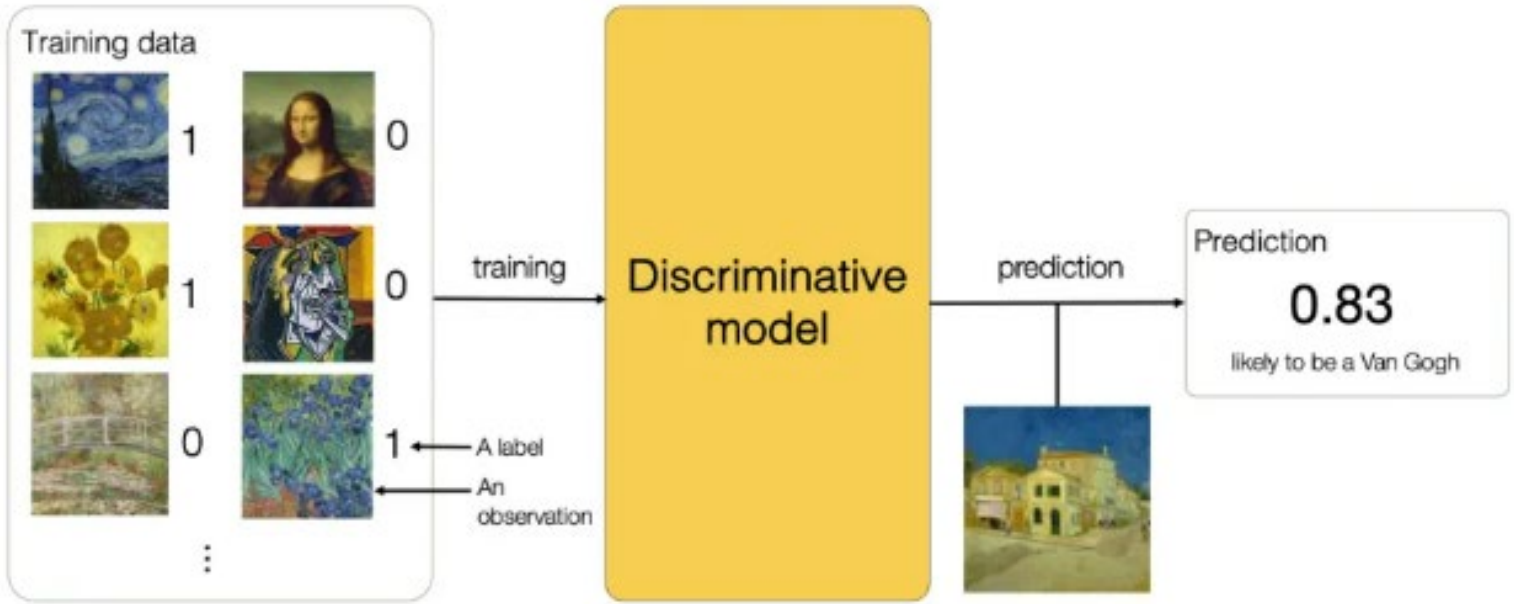
Generative Models (GM)

- GMs are probabilistic rather than deterministic
  - Reason: *sample* many variations of the output
    - If probabilistic, a random component influences the individual samples generated by the model.
  - Instead of: get the same (or approximated) output every time



Generative vs. Discriminative Modeling

- Discriminative:
  - Each observation in the training data has a *label*
  - For a binary classification, 1 – for Van Gogh, 0 – for non Van Gogh



Generative vs. Discriminative Modeling

- Formally:
  - Discriminative modeling estimates $p(y|x)$
    - Model the probability of a *label* y given some observation x
  - Generative modeling estimates $p(x)$
    - Model the probability of observing an observation x. Sampling from this distribution allows us to generate new observations.
  - Conditional Generative Models: $p(x|y)$
    - The probability of seeing an observation x with a specific label y.
    - For example, the dataset is about different types of vehicles, tell the model to generate an image of an 18-wheeler.

Generative Models (GM): Framework

- We have a dataset of observations X
- We assume that the observations have been generated according to some unknown distribution, P_{data}
- We want to build a GM P_{model} that mimics P_{data}. If we achieve this goal, we can sample from P_{model} to generate observations that appear to have been drawn from P_{data}
- The ideal for P_{model} is:

Generative Models (GM): Framework

- The ideal for P_{model} is:
  - Accuracy:
    - If the accuracy of P_{model} is high for a generated observation, it should look like it has been drawn from P_{data}
  - Generation
    - It should be possible to easily sample a new observation from P_{model}
  - Representation
    - It should be possible to understand how different high-level features in the data are represented by P_{model}

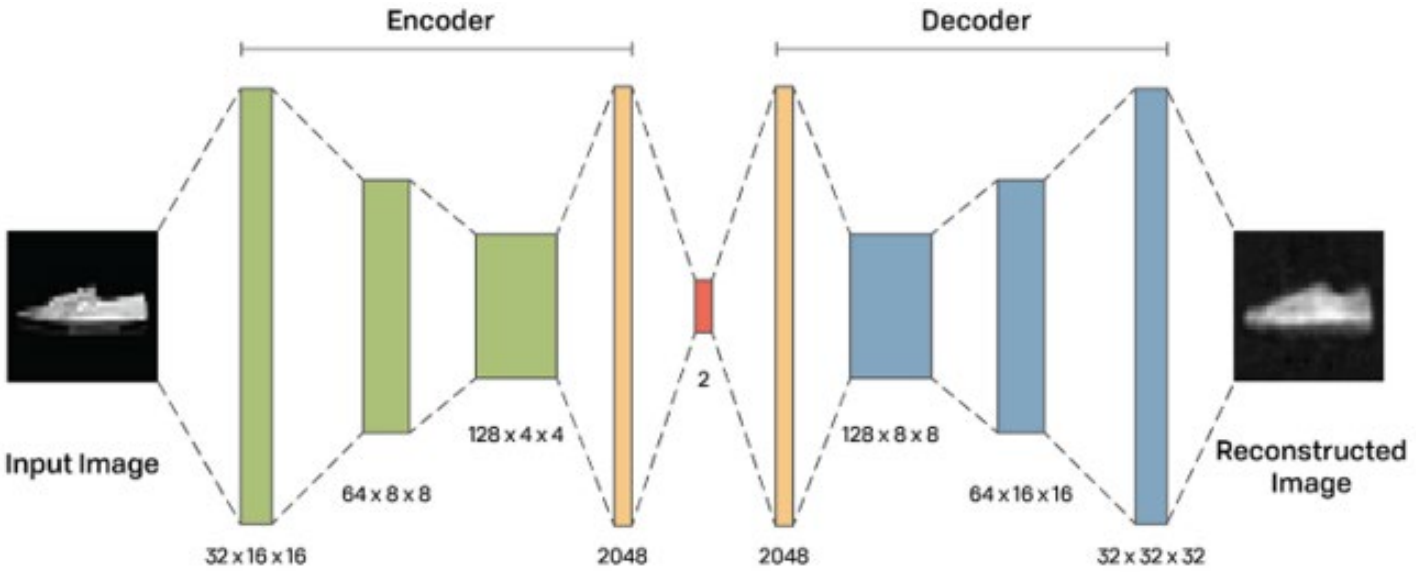
Generative Models (GM): Representation

- Representation of high-dimensional data (representation learning):
  - Instead of trying to the model high-dimensional data sample space directly, we describe each observation in the training set using lower-dimensional **latent space** and map it to a point in the original domain.

  - "Short definition": each point in the **latent space** is a representation of some high-dimensional observation

Generative Models (GM): Autoencoder

- Autoencoder:
  - Shor definition: A neural network that is trained to perform the task of encoding and decoding an item
    - The output should be as close to the original item as possible

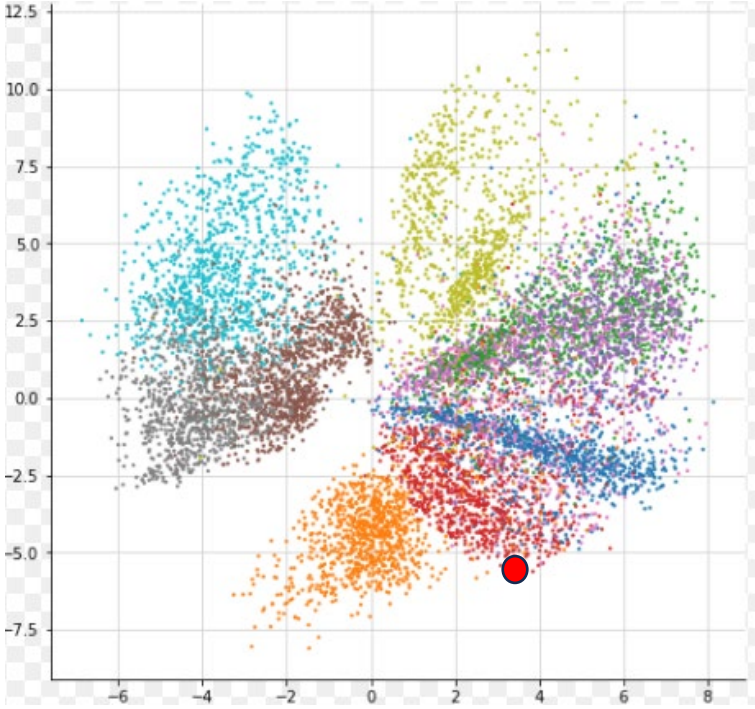


Generative Models (GM): Autoencoder

- Why reconstruct images that are already available?
  - Sampling from the latent space -> creation (generation) of new unseen image
- Embedding:
  - Compression of the original image into a lower-dimensional latent space
  - Done by the **Encoder**
- **Encoder**
  - Main job: to take an input image and map it to an embedding vector in the latent space

Generative Models (GM): Autoencoder: **Encoder**

- Encoder architecture:
  - Define the Input layer of the encoder (the image)
  - Stack Conv2D layers sequentially on top of each other
    - Remember the convolutional process for images:
      - Some pixel data will be lost due to filtering!
  - Flatten the last convolutional layer to a vector
  - **Connect this vector to the <u>2D embedding</u> with a Dense Layer**
    - This allows to plot the latent space



For example: a sample can be: [-2.5,-6.0]

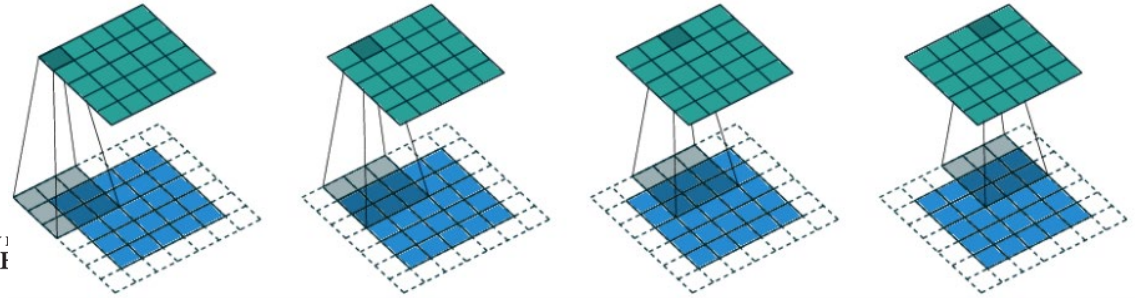
Generated image mapped to latent space

Generative Models
Autoencoder: Decoder

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI

Generative Models (GM): Autoencoder: Decoder

- The decoder is a mirror image of the encoder:
  - Instead of convolutional layers, the decoder uses **convolutional transpose layers**.
  - Convolutional transpose layers:
    - In **regular convolutional layers**, the resulting output is reduced due to the filtering processes, remember key terms:
      - For example, an input image of 32x32x1 with stride = 2 will produce a 16x16x1 output
      - *Stride: step size used by the layer to move the filters across the input. By Increasing the stride, the size of the output (tensor) is reduced.
      - *Padding: “pads” the input data with zeros (zero padding) so that the output size from the layer is exactly the same as the input size when strides = 1

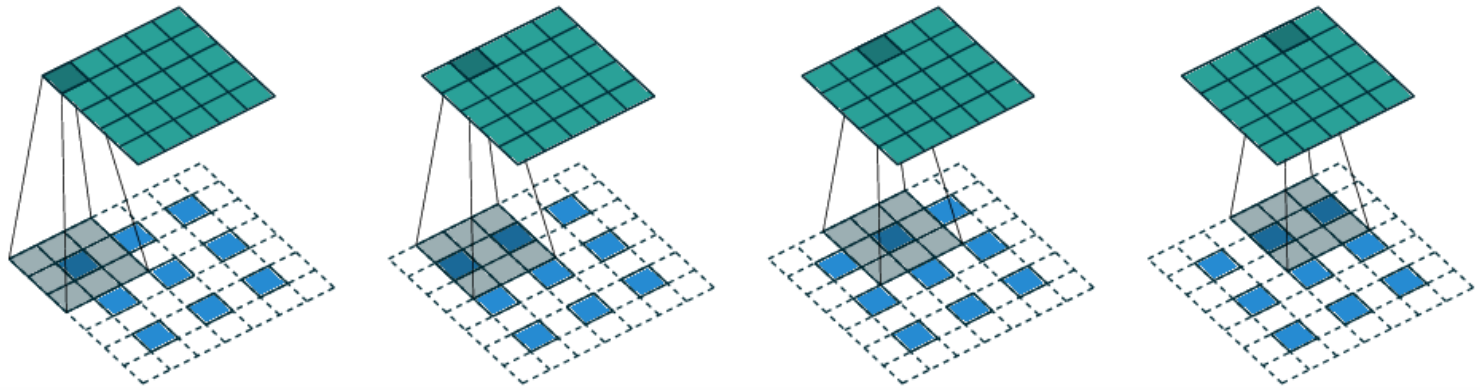


A 3x3x1 kernel (gray) being passed over a 5x5x1 input image (blue), with padding = ‘same’ and strides = 1 to generate the 5x5x1 output (green).
Padding = ‘same’ allows to easily keep track of the size of the tensor.

Generative Models (GM): Autoencoder: Decoder

- In the case of **convolutional transpose layers**:
  - Use the same principle as a standard convolutional layer, but setting **strides = 2** doubles the size of the input tensor in both dimensions.

  - In convolutional transpose layers, the strides parameter determines the internal zero padding between pixels in the image



A 3x3x1 filter (gray) is being passed across a 3x3x1 image (blue) with strides = 2 to produce a 6x6x1 output sensor (green).

The **Conv2DTransponse** method in Keras allows the conv. tranpose operations until the image reaches its original dimensions. The 0-padding method in transpose "adds" extra space according to the stride parameter.

Generative Models (GM): Autoencoder: Joining the Encoder and Decoder

- To train the encoder and decoder simultaneously we need to define a model that will represent the flow of image
  - Flow: Encoder and back out through the decoder

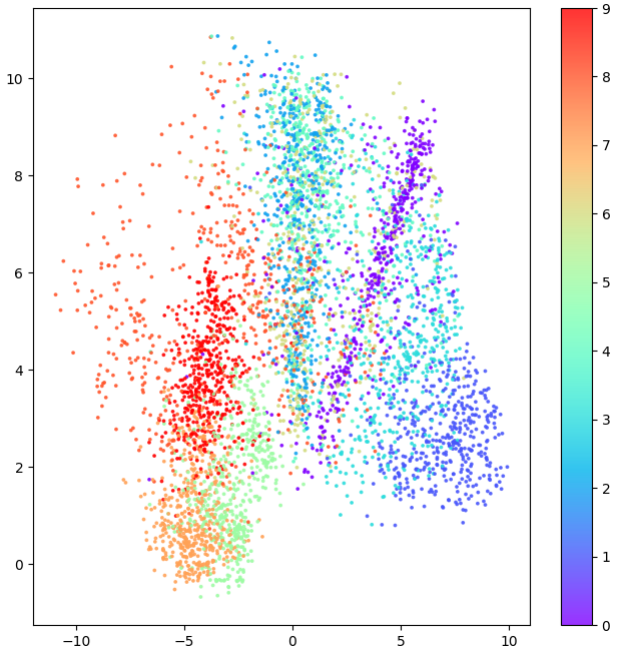
- In Keras, the flow can be specified:
  - Output from the autoencoder is:
    - The output from the encoder after it has been passed through the decoder.

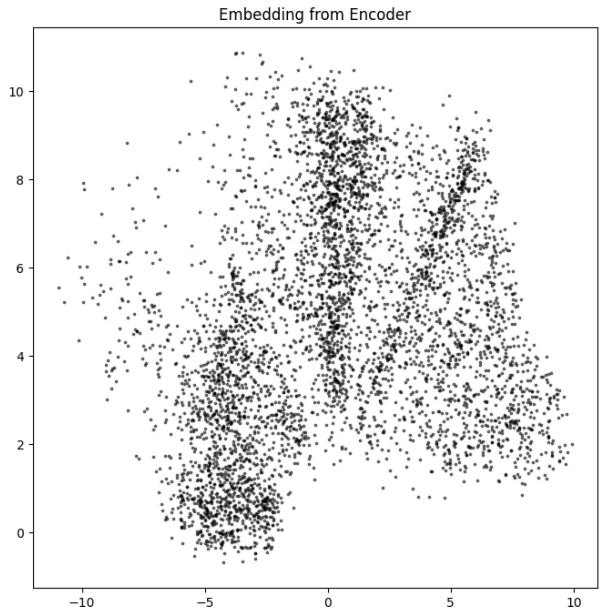
- `autoencoder = Model(encoder_input, decoder(encoder_output))`

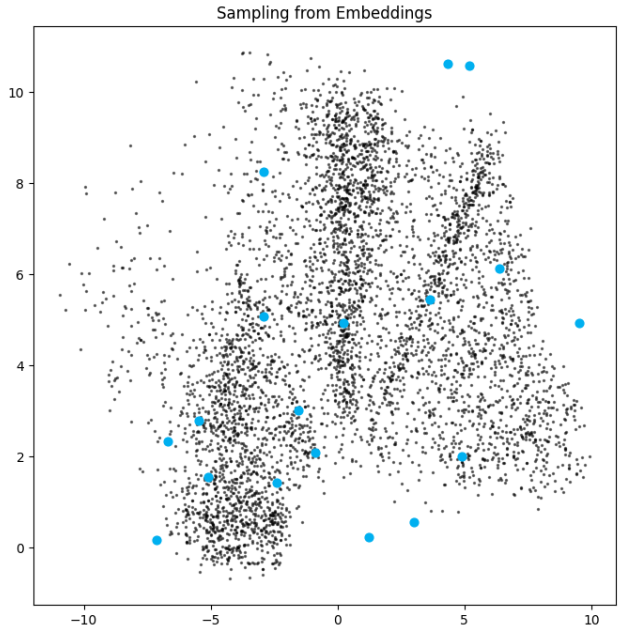
- Model takes an image and passes it through the encoder and back out through the decoder to generate a reconstruction of the original image.

Generative Models (GM): Decoder

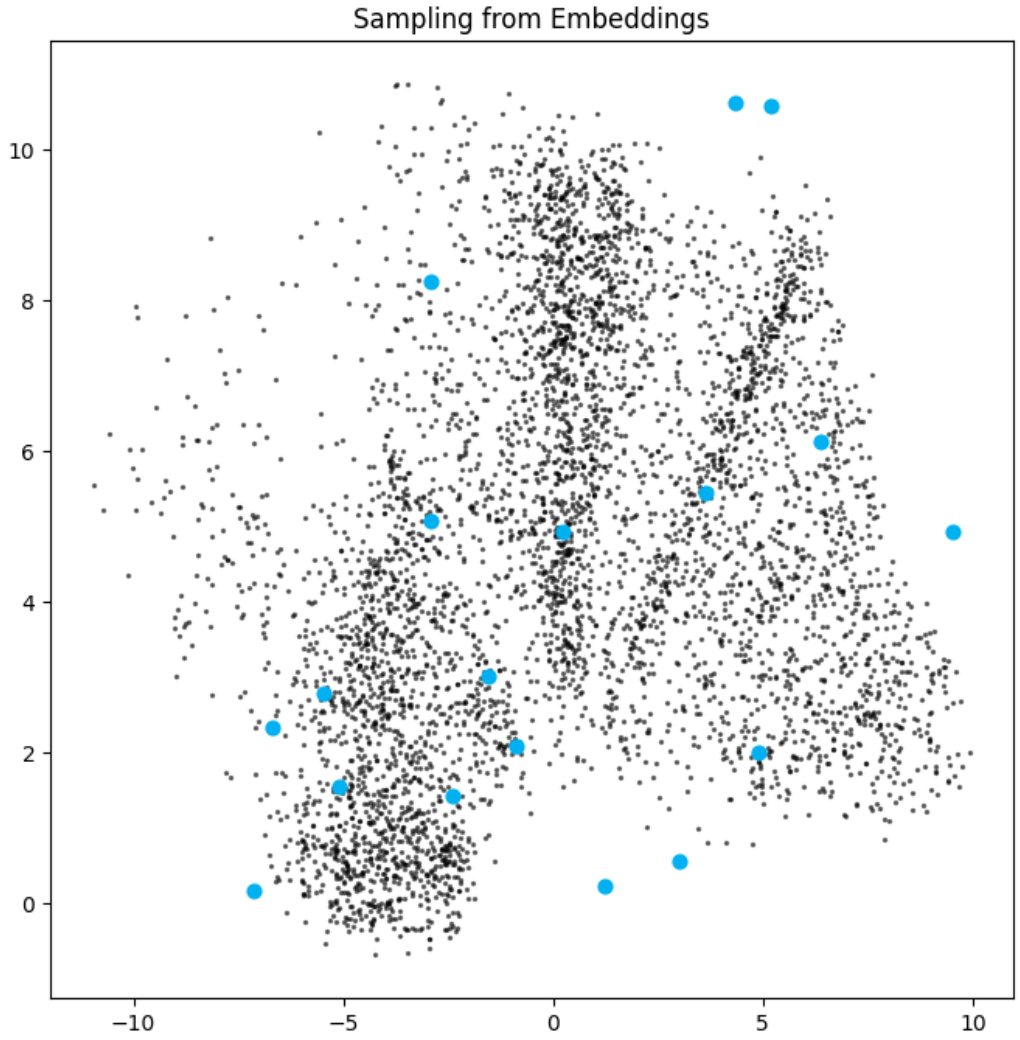
- Generating new images
  - We can generate novel images by sampling some points in the latent space and using the decoder to convert these back into pixel space.







Generative Models (GM): Decoder



Example

On Canvas: Download Autoencoder_test.py

Experience Replay

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI

Experience Replay (ER)

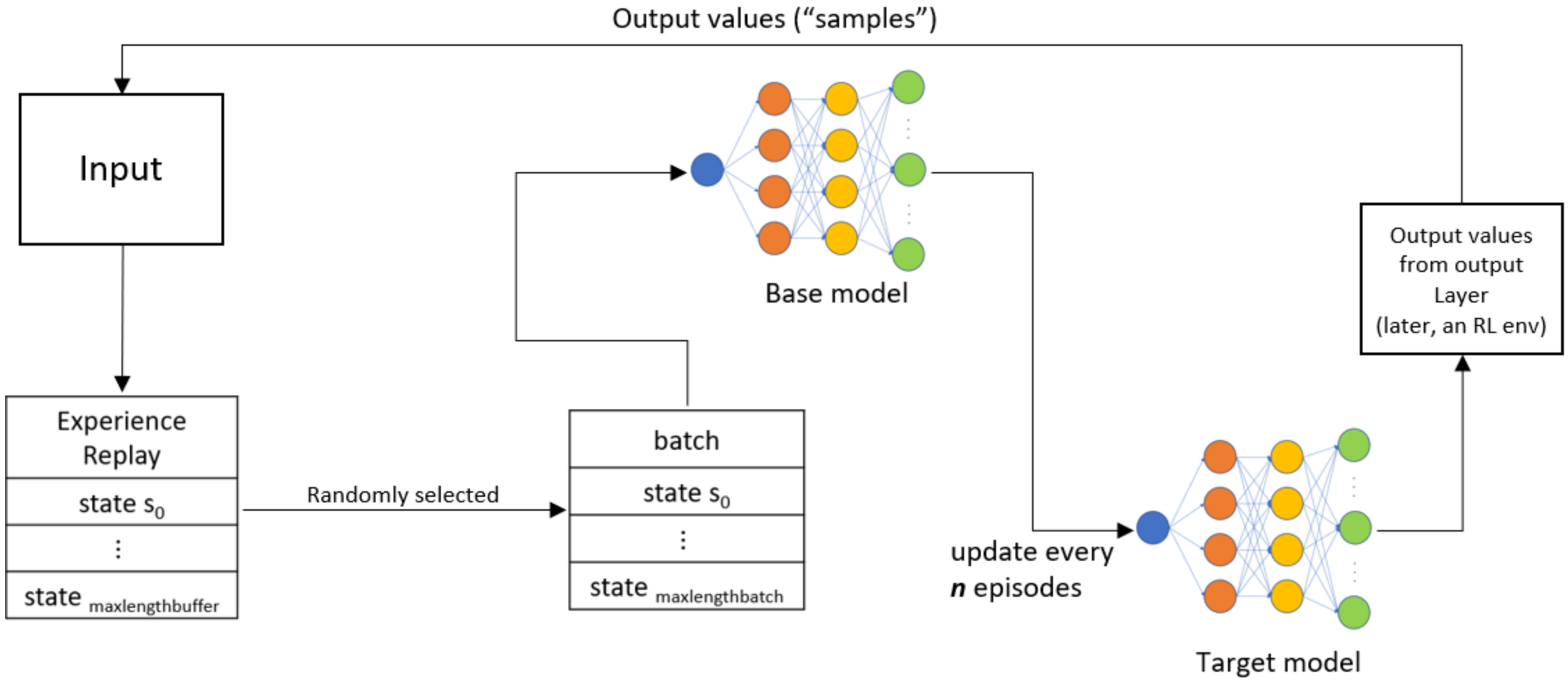
- Implemented to increase the efficiency of a training model
  - Samples are randomly selected and saved at each step during an episode
  - Samples are randomly selected to train the model
  - Samples come from the output layer of the NN
    - This will change to output of an RL environment

- ER
  - Smoothens fluctuations
  - Reduces correlation of samples

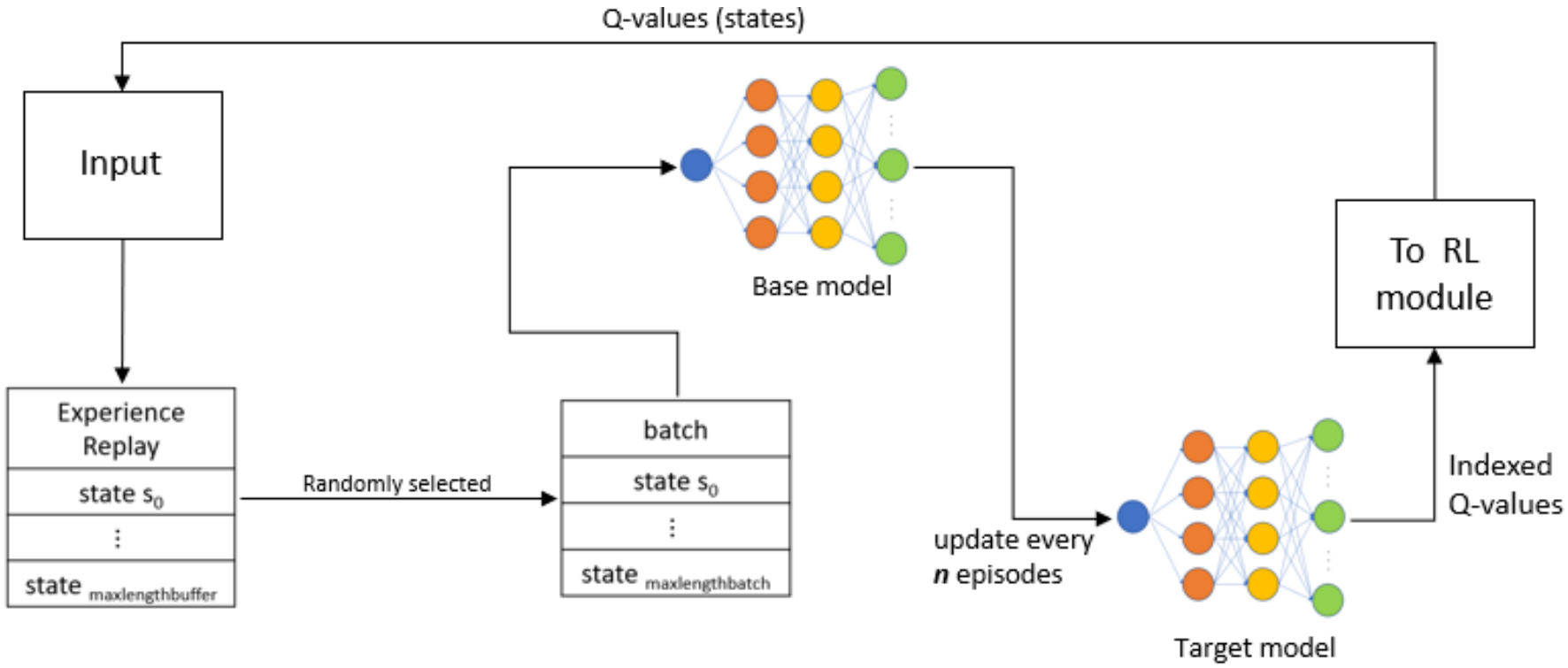
- Some terms:
  - Outputs of the output layer of the NN will be “samples”
  - Buffer: we will randomly save the samples in a buffer
  - The new inputs to the NN will come from the buffer
  - The buffer is now a Replay Buffer

  - Replay buffer: Saves “experiences”
    - “Table” that saves random samples to be passed for training
    - Size of the replay buffer varies. It will be a hyperparameter

  - Training by batch:
    - Now that we will be using a replay buffer to train, we can make a smaller group of samples. This will be a batch.
    - The NN will use the samples from the batch to train.

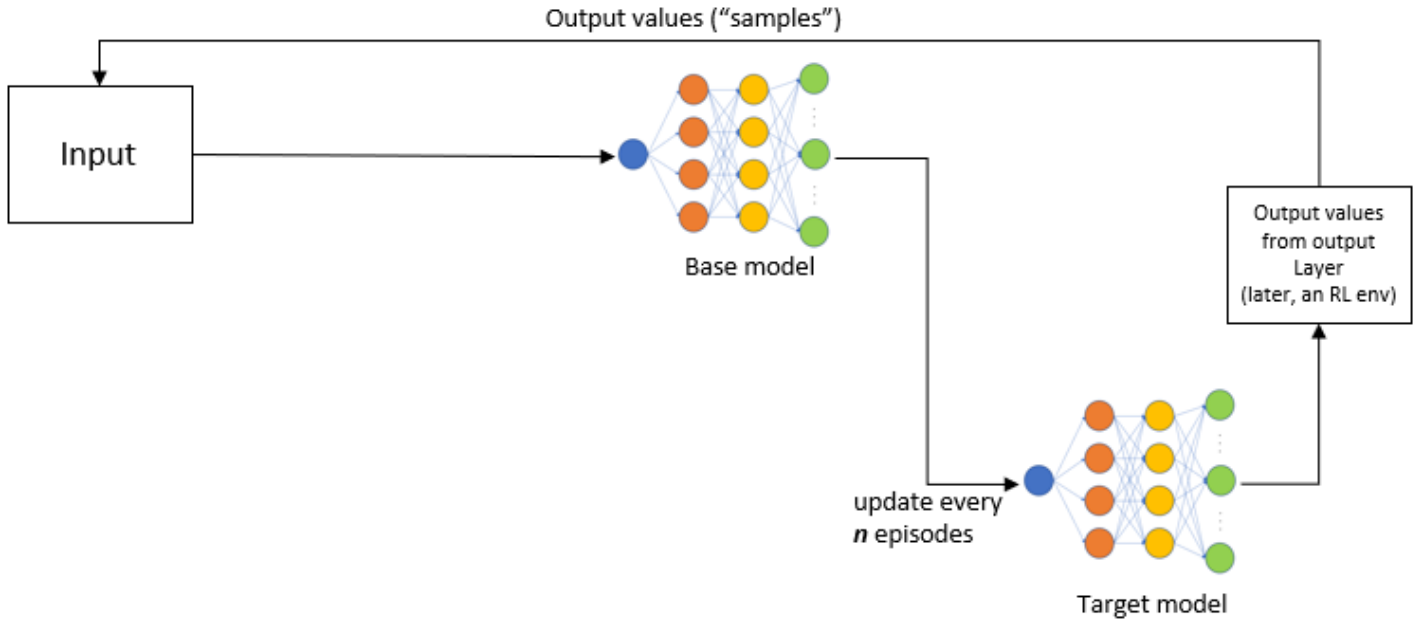

Output values (“samples”)
Input
Base model
Output values from output Layer (later, an RL env)
Experience Replay
state s_0
state $_{maxlengthbuffer}$
Randomly selected
batch
state s_0
state $_{maxlengthbatch}$
update every *n* episodes
Target model

- We will use ER in NN for:
  - **Approximation** of values for:
    - Q-values: indexed.
    - This is Deep Reinforcement Learning



- Training by batch vs single input:
  - The batch contains random experiences
  - The experiences are passed to the NN
  - The NN can continuously be fed from the batch

- Single input
  - One input at a time
  - Weight & bias corrections (optimization updates) are done per input
    - It Takes longer for the NN to update



- Hyperparameters so far:
  - Number of episodes (in NN: “epochs”)
  - Number of steps (In RL, for example: X steps = 1 episode)
  - Epsilon (tied to explore/exploit):
    - Epsilon decay
    - Max & min epsilon
  - Dropout
  - Length of the replay buffer
  - Length of batch (mini-batch)
- Metrics:
  - Accuracy
  - Loss

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Epsilon in DRL

DRL

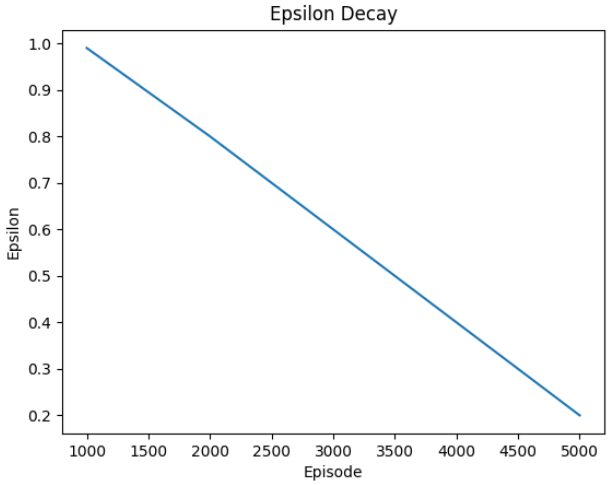
- So far, a DRL model has multiple hyperparameters:
  - Epochs (episodes)
  - Dropout
  - Learning rate
    - Tied to optimization method
  - Epsilon
    - Linear decay
    - Exponential decay

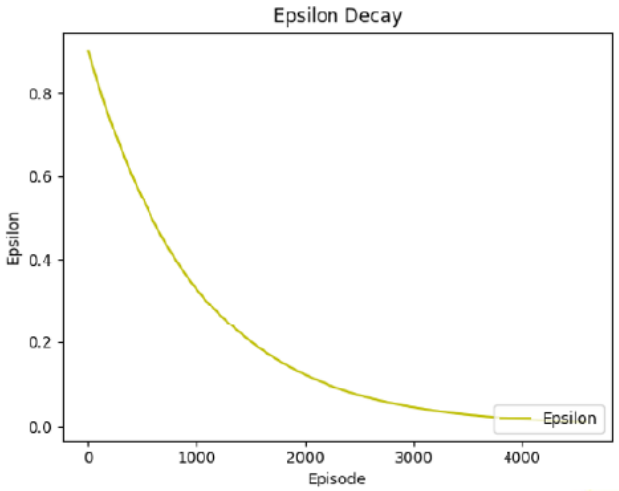
Gathering information

- The agent needs to gather information:
  - We control this by using:
    - Exploration – Exploitation methods
  - Upper confidence bound:
    - Looks for “uncertainty”
      - “Locks in” a Q-value and explores around it
  - Epsilon-greedy (ε-greedy)
    - Linear decay
    - Exponential decay
  - Other forms:
    - sinusoidal -> depends on the environment
    - Softmax = Not to be confused with Softmax activation method

ε-greedy

- With ε-greedy, the decay can be controlled:
  - Linear
    - Decay occurs in a linear form
    - Exploration – exploitation is “constant”
    - Variation:
      - Decay rate (decay step)
  - Exponential
    - Decay occurs in an exponential form
      - Exploration – exploitation is variable
        - Explore at the beginning, exploit at the end
      - Variation:
        - Decay rate (decay step, & explore, exploit rate)





Environments

- For example:
  - Slot machine:
    - One arm
    - Different combinations to get a reward
    - One action – get rewarded
    - One – arm bandit
      - Multiple – arm bandit
  - How to explore – exploit the environment:
    - One arm, or multiple arm
    - Linear decay
    - Exponential decay

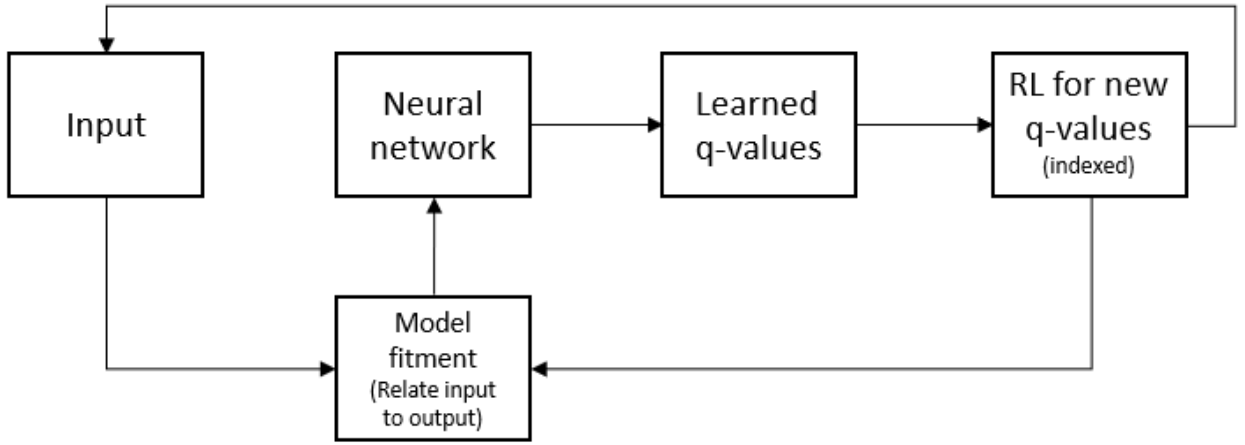
THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

DRL, generalization, and DQN

Deep Reinforcement Learning (DRL)

- Originates from the combination of:
  - RL using a value-based technique: Q-Learning
  - A deep neural network

- It can be defined as:
  - Implementation of deep neural networks to approximate the components in RL, such as Q-values.



DRL

- With DRL:
  - Q-learning:
    - Learning is done through the approximation of Q-values instead of prediction

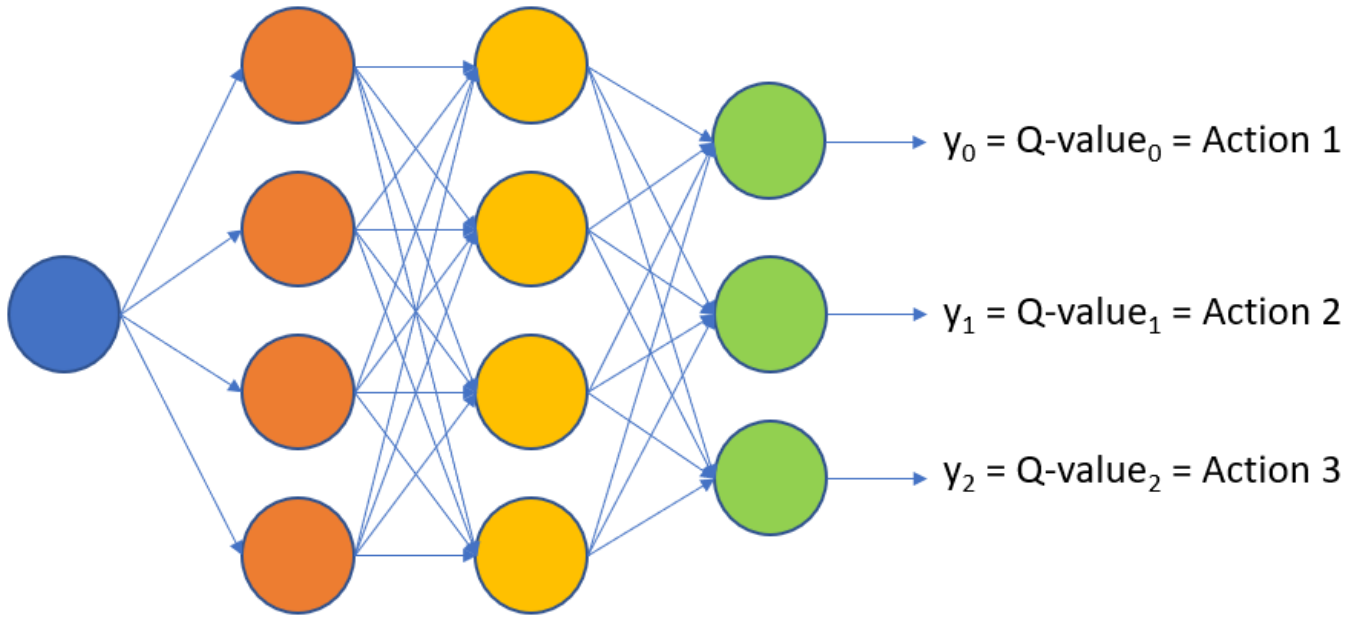
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right)\right]$$

Approximated through NN

- Now, the RL agent queries the NN for Q-values so that an action can be executed
- Output of the NN are indexed

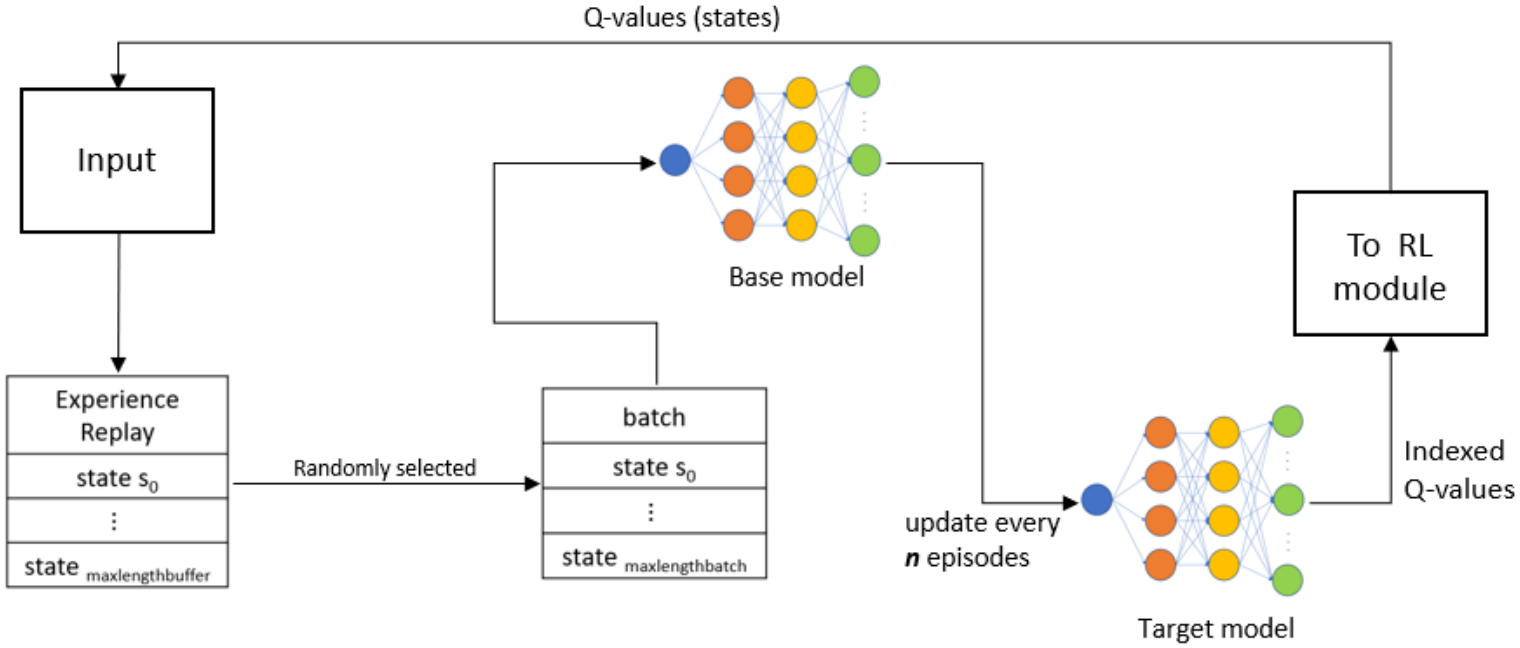
DRL

- Indexed NN output:
  - The output with the highest output value is passed to the RL agent



DRL -> DQN

- With NN approximation, a Q-table can be disregarded
  - Everything is now coming from the NN
  - **This is a Deep Q-network**
- However, a “warm-up” period is or can be needed to discard unnecessary experience



DQN

- Some cautions with DQN:
  - Since we are working with large state spaces:
    - Dimensionality: the number of states is very large
  - To address this:
    - Use Experience Replay
      - Prioritized experience replay
      - Importance factor

DRL

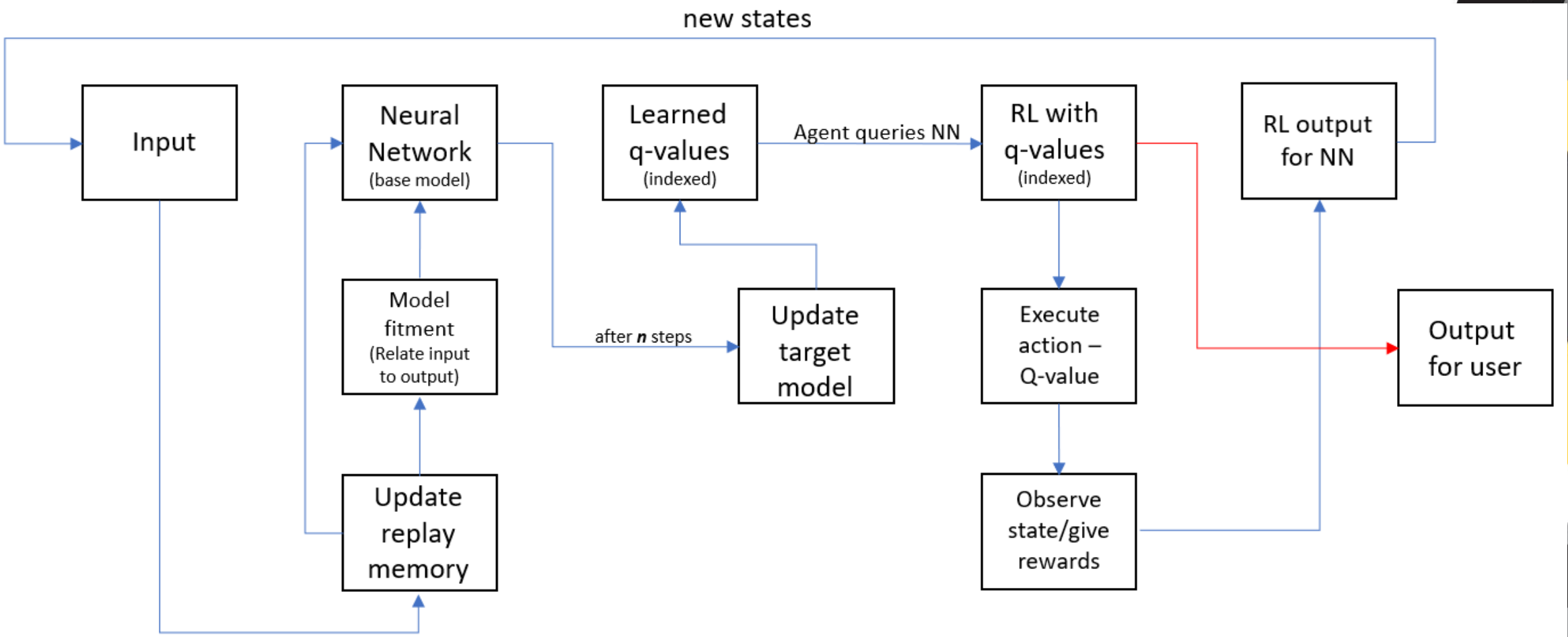
- Generalization:
  - Two definitions:
    - The “problem” of generalization
    - A generalized model
  - The “problem” of generalization:
    - When a model is not able to approximate (or classify) because of **overfitted** values:
    - Overfitted values:
      - False values for Accuracy and Loss:
        - Very high accuracy reached, very fast
        - Very low loss reached, very fast

DRL

- A generalized model:
  - A single model that can be implemented in different environments:
    - This means:
      - No overfitting
      - Good accuracy and loss
  - A generalized model can still output good values even when the input state was never seen before (or during training)

- The main goal of a DRL model is to be able to generalize

DQN flowchart simplified with ER

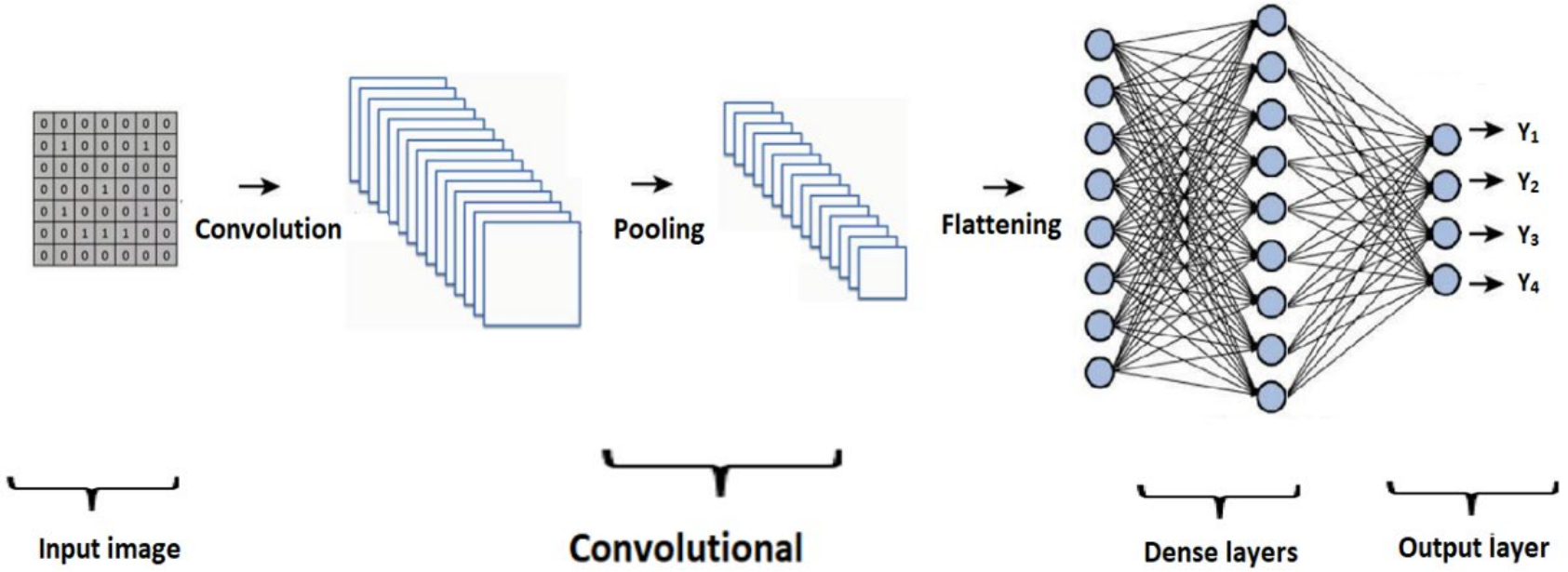


THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

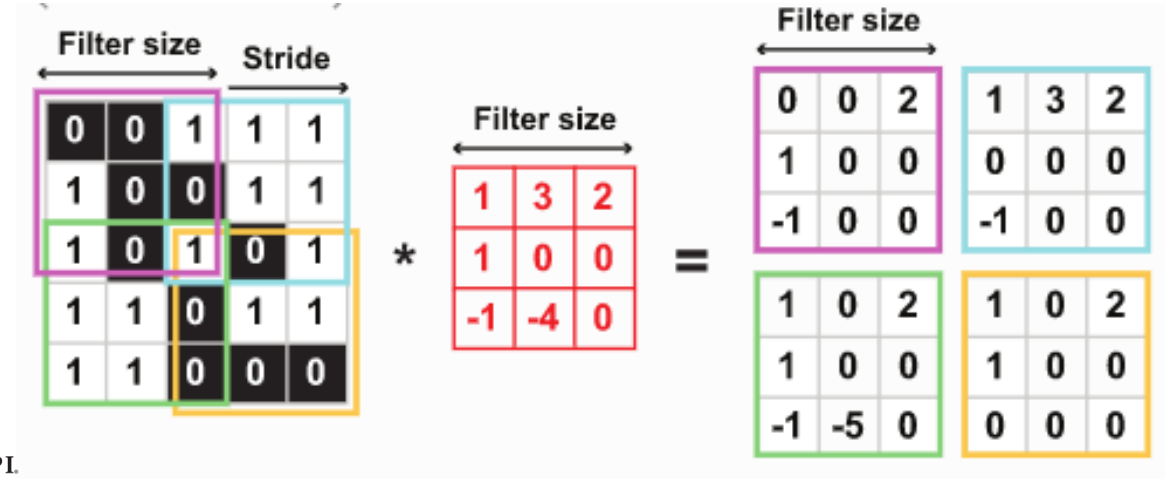
Convolutional layer & GPUs

Convolutional layer

- The convolutional layer prepares the input data for the fully connected neurons (hidden layers)
- Procedures happening the convolutional layer:
  - Filtering
  - Pooling
  - Flatten &

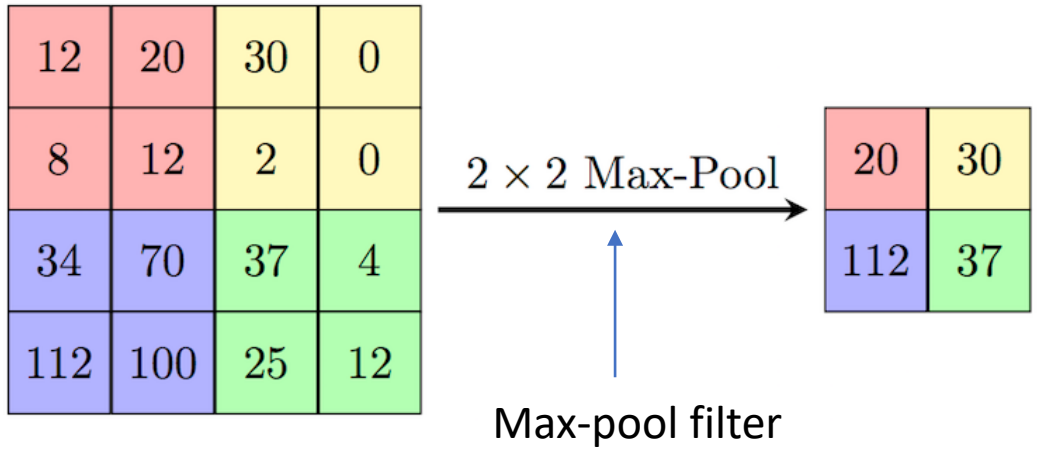


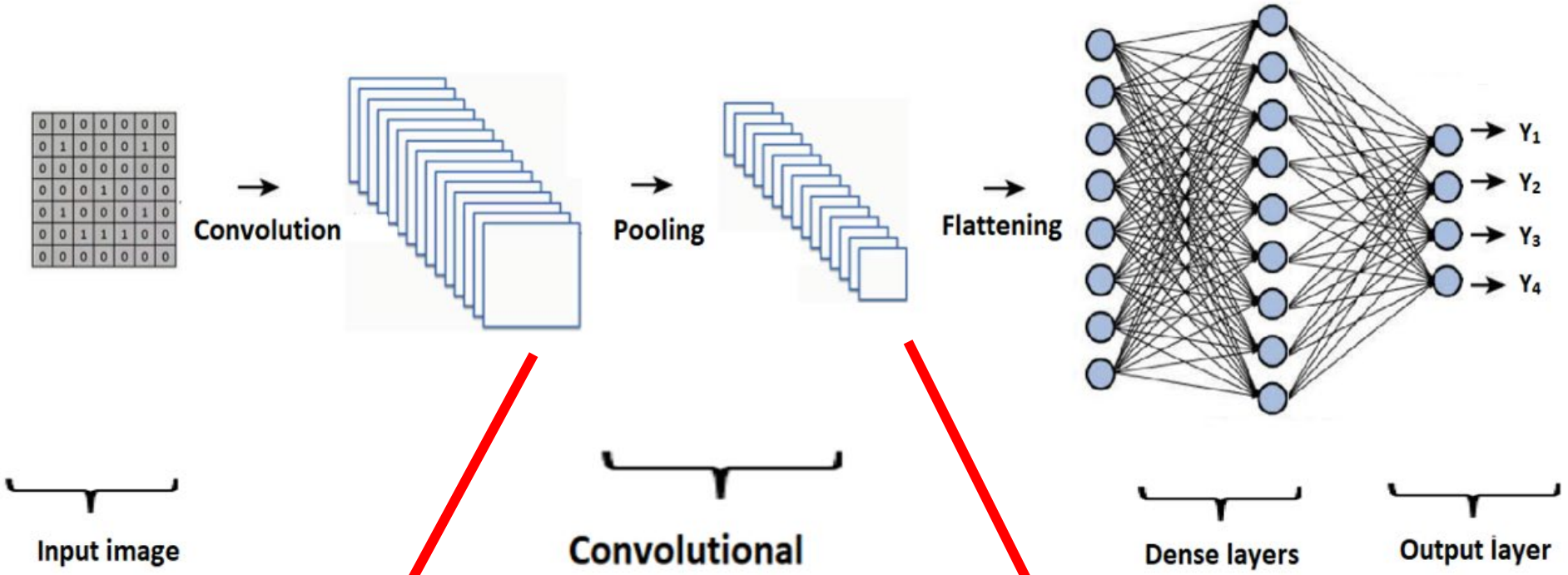
- Filters (kernel):
  - In charge to detect features
    - In images:
      - Can detect for example, lines, corners
      - Can also be specific: eyes, hair, etc.
  - Is a matrix of m * n. Size of the matrix depends on the application
    - Initialized randomly
    - Stride: how many “cells” the filter moves for a dot product
  - Operation:
    - A dot product operation is executed between the input data and the filter

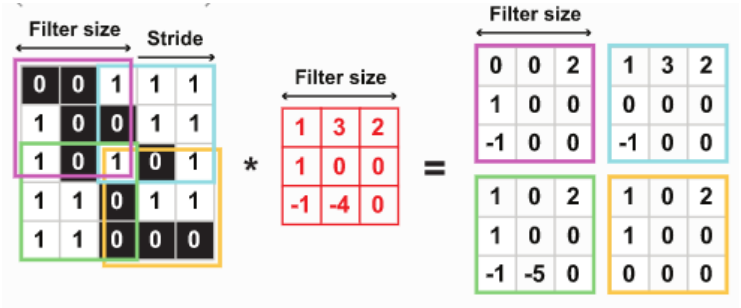


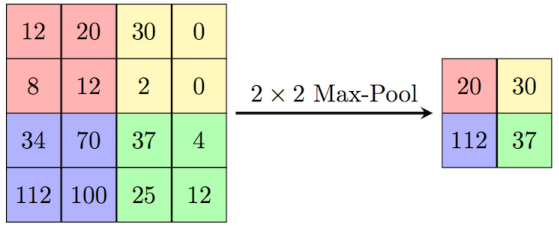
- Pooling:
  - To reduce dimensionality
  - Eliminates noise from data

  - Max pooling: after the dot product of the filter and input is executed
    - A max pooling filter (of m*n) selects the max value and creates a new max pool.

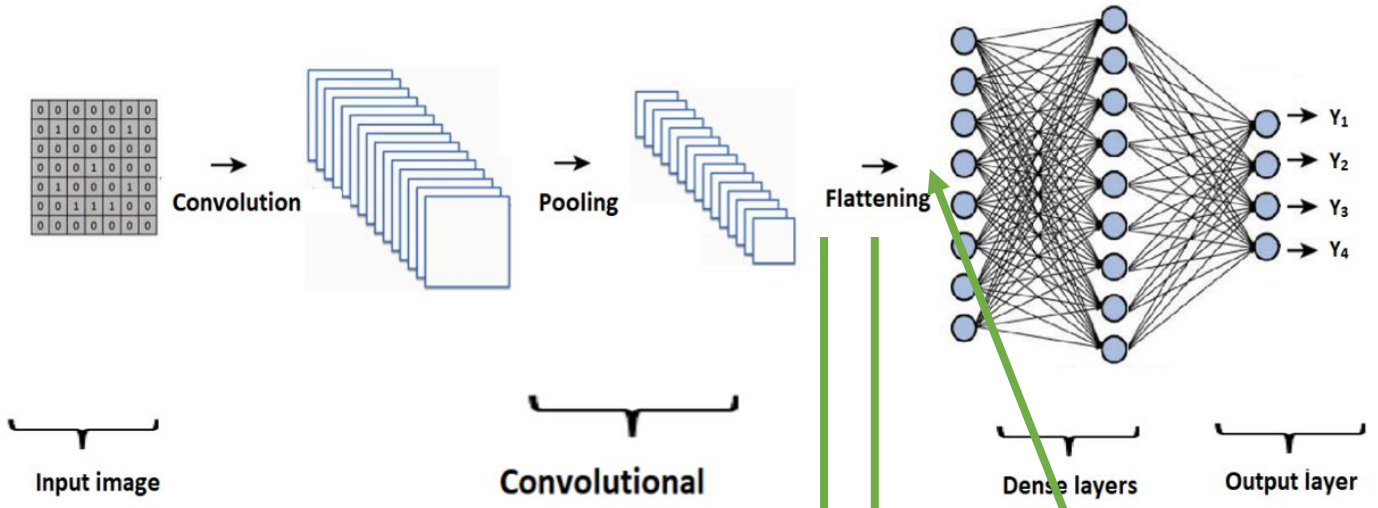



Convolution
Pooling
Flattening
Y1
Y2
Y3
Y4
Input image
Convolutional
Dense layers
Output layer


Filter size
Stride
Filter size
Filter size


2 × 2 Max-Pool

- Flatten & Dense:
  - Flatten: For the neurons to perform their own operations, the input data needs to be an appropriate format:



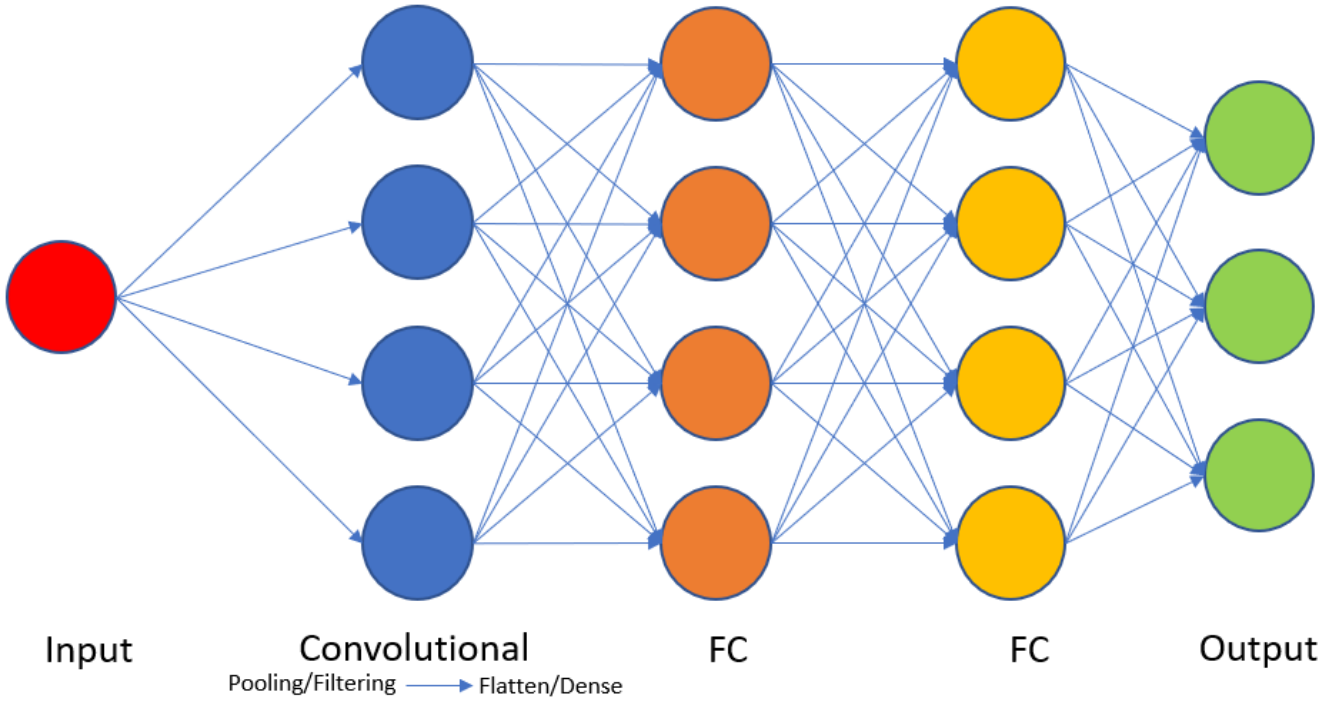
The output will be the classification
based on the features extracted through the previous layers and their different filters (classification: outputs are mapped to labels)

The vector (tensor) is passed as input of the FC

The output if the max-pool is a matrix
Can be of n-dimensions

Flattening: "Converts" n-dimension into a "readable" vector (tensor)

- Dense:
  - Method to create a fully connected neural network
  - Specifies the number of neurons per layer and activation method



- How is a CNN processed when GPUs are available?

GPU ARCHITECTURE

Streaming Multiprocessor (SM)

Many CUDA Cores per SM

Architecture dependent

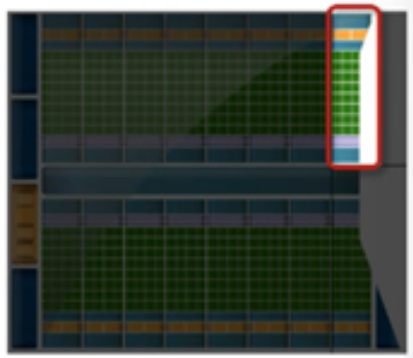
H100 SM has 128 cores

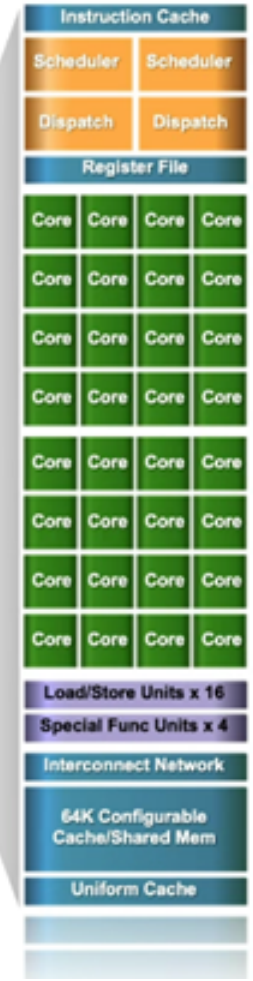
Special-function units

cos/sin/tan, etc.

Shared mem + L1 cache

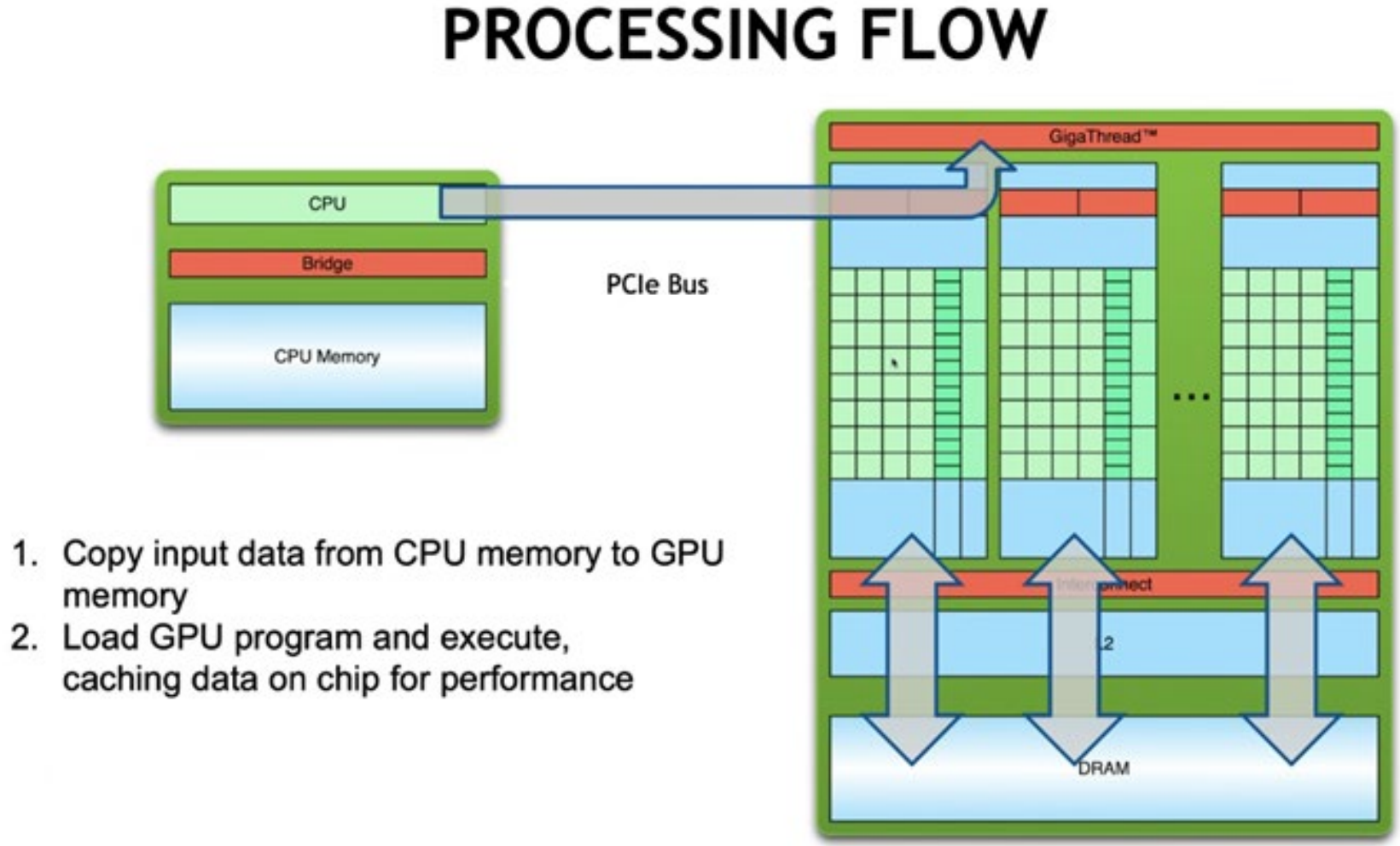
Thousands of 32-bit registers



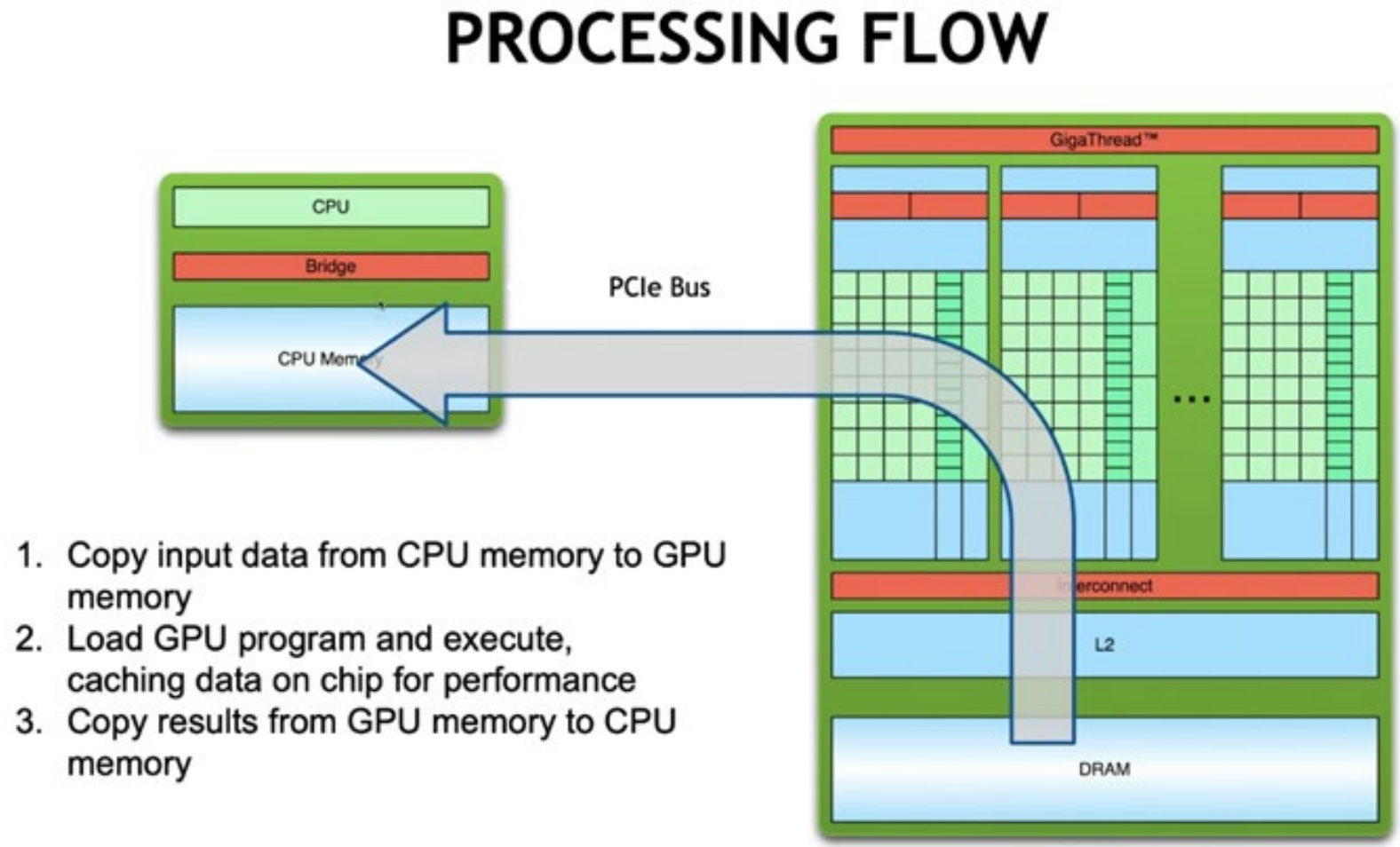


H100 PCIe has a total of 14,592 cores

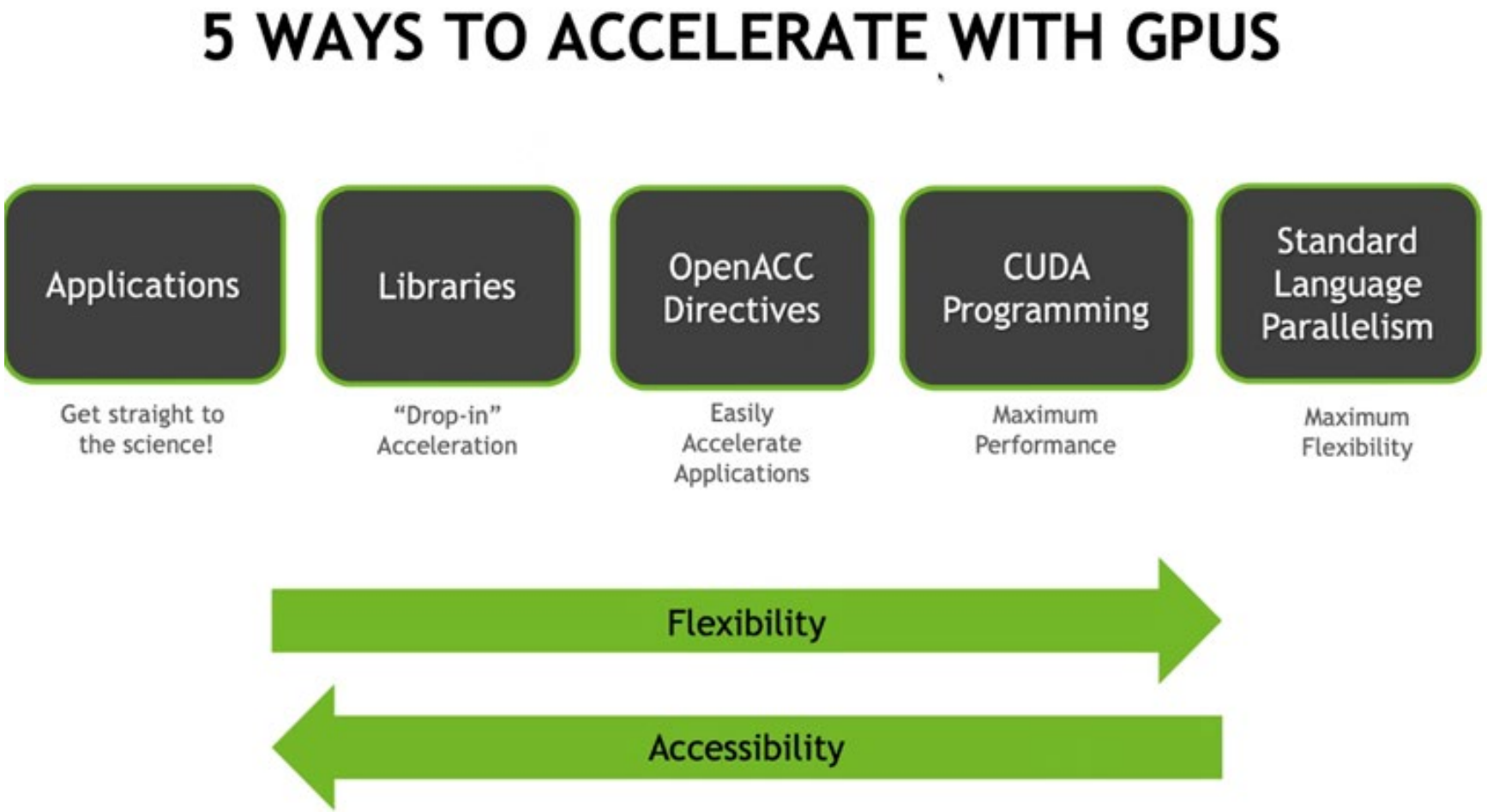
- How is a CNN processed when GPUs are available?



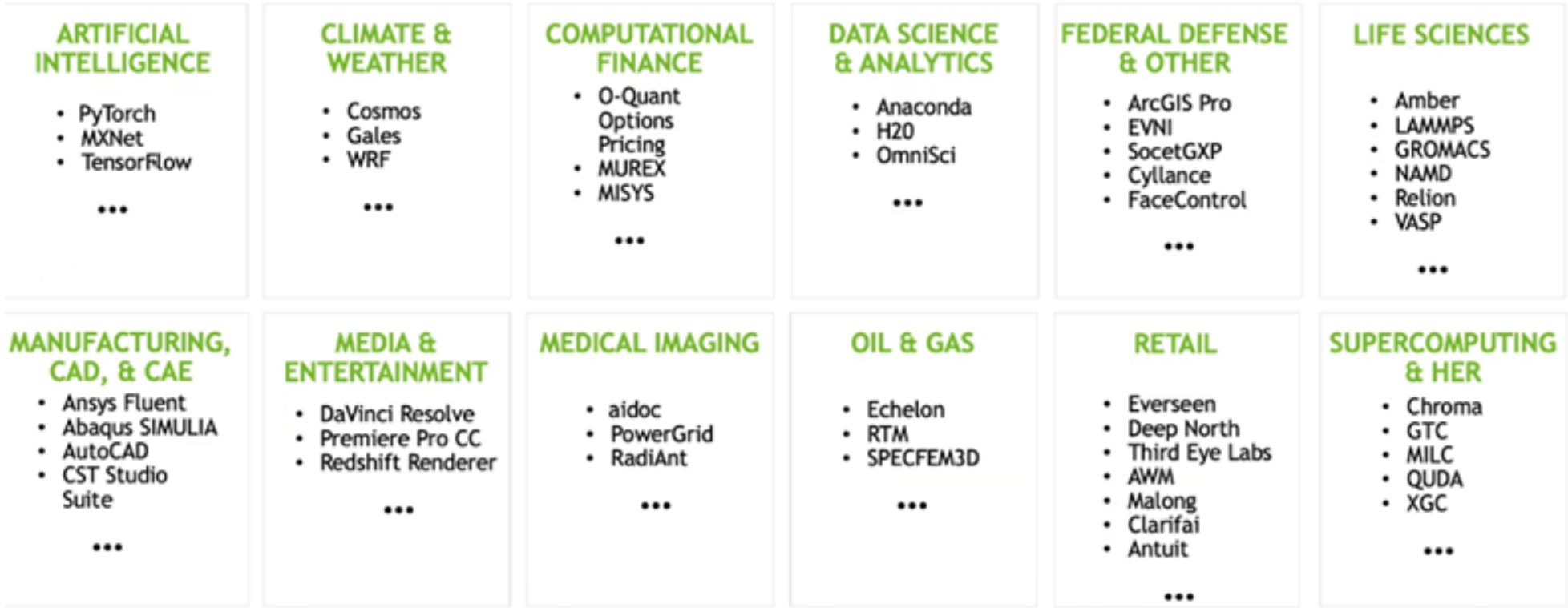
- How is a CNN processed when GPUs are available?



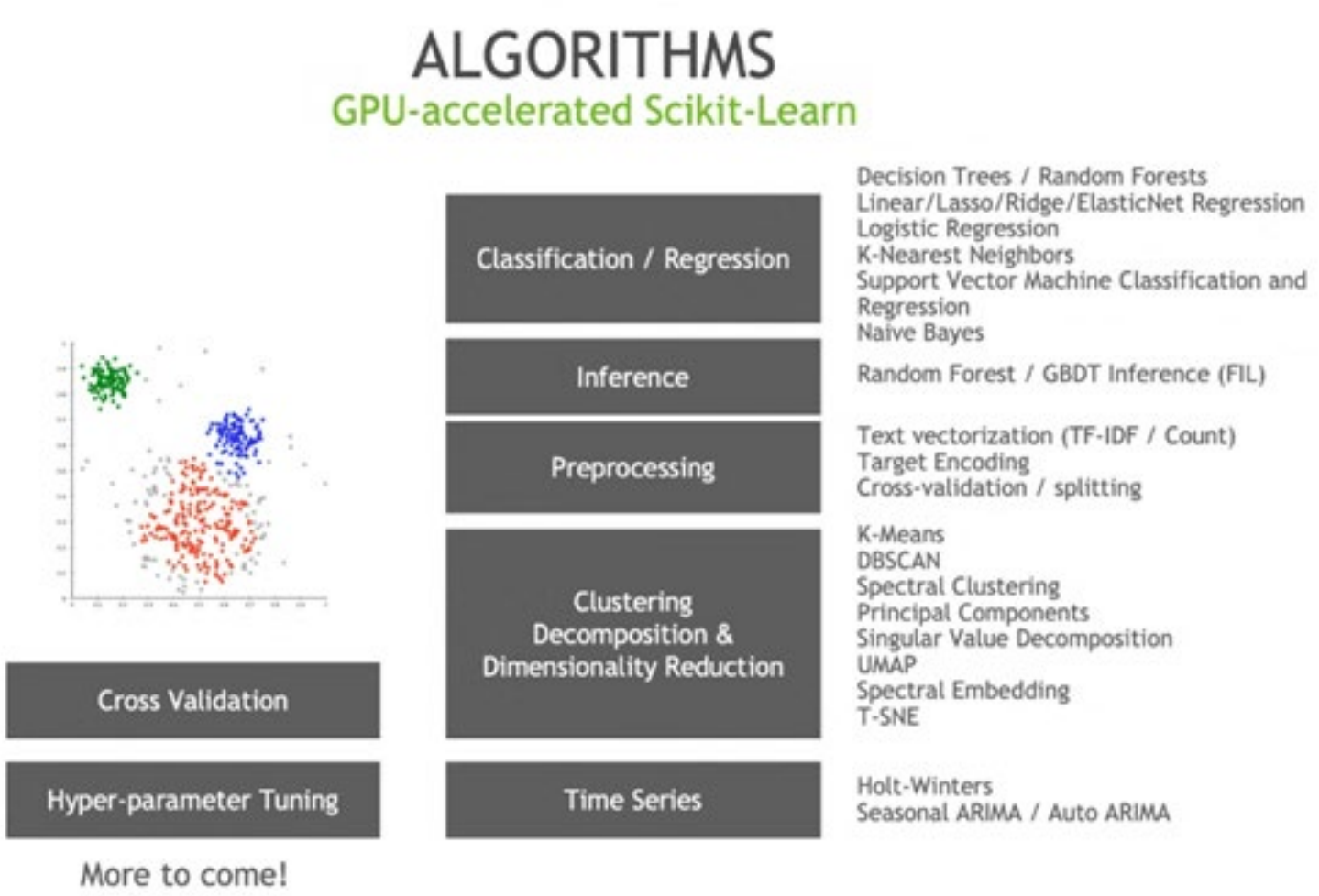
- How is a CNN processed when GPUs are available?



- How is a CNN processed when GPUs are available?



- How is a CNN processed when GPUs are available?



THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®

Challenges in RL
Environment: size, action space, and state space

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI

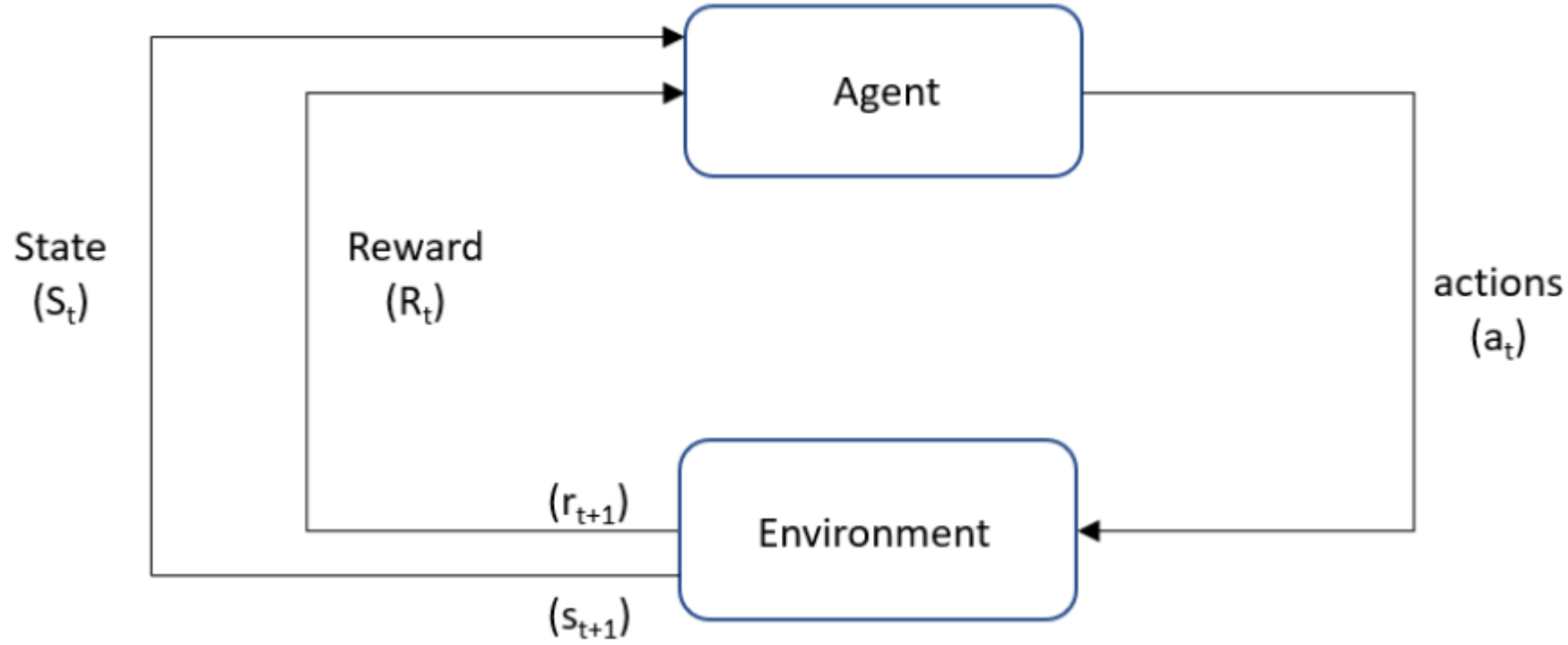
Challenges in RL

- In Q-learning, a Q-table is implemented:
  - From the robot in the grid example:
    - 3x3x4 = 36 total states -> The robot can be at any location in any orientation
    - 4 different states per 3x3 grid
      - Actions = Left, right, fwd. Location = (orientation) up/down/left/right
      - The state of the world is the robot's position + orientation

source	Target		
			
		(1,1, facing up)	
		(0,1,facing up)	

Challenges in RL

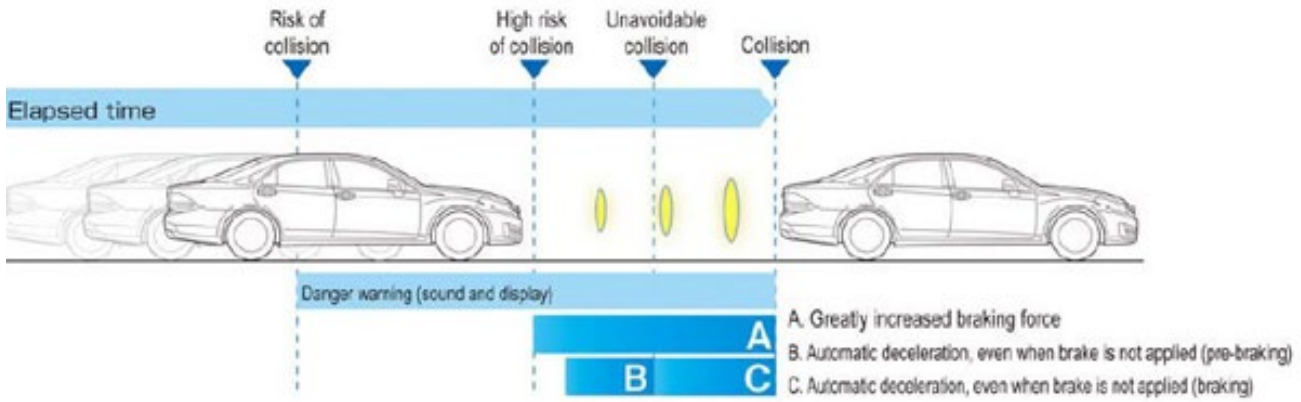
- Moving from (0,1) to (1,1) and obtaining a Q-value:
  - Select an action from the pool
  - Execute selected action
  - Approximate a new q-value: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q\,(S_{t+1}, a)\right]$
  - Save Q-values in table:
    - Here, the complexity increases:
      - Q-Table needs to contain:
        - Combination of: actions, state, and what q-values were given for the combination of (action, state) pair
        - The Q-table is now an n-dimensional list.



- State: What it's observed
- State space: All possible states for a system
- Action space: set of actions. Has to be finite.

Challenges in RL

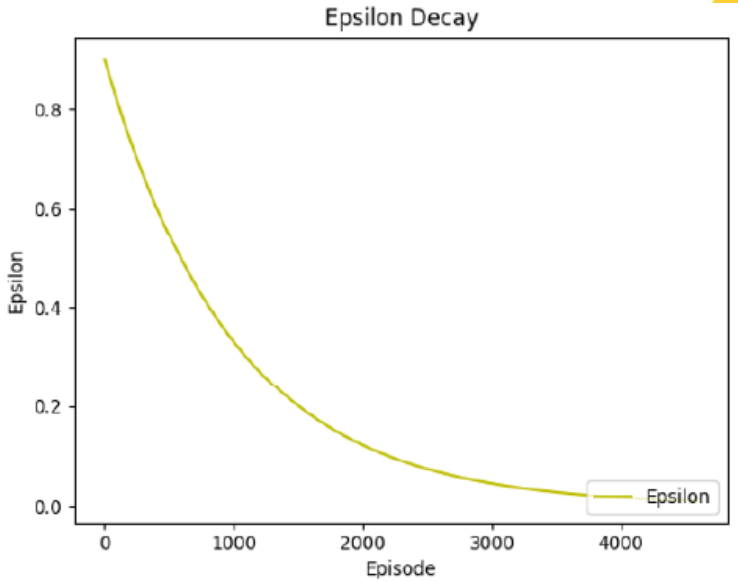
- To train a RL agent to create a model that will predict stopping parameters to prevent side collision:

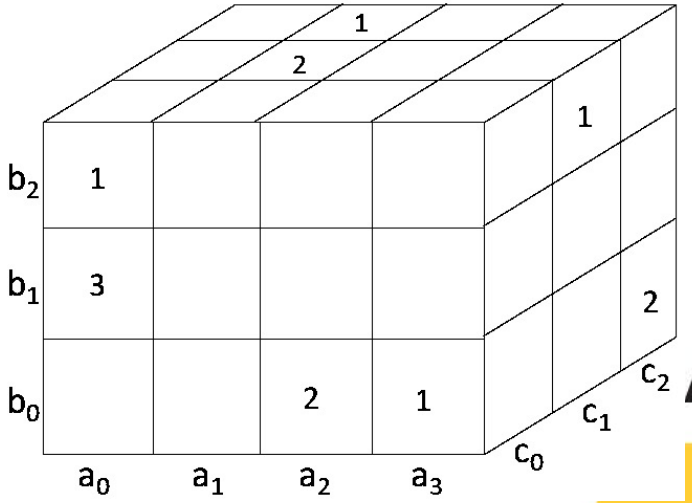


Some Actions	Other factors
Increase breaking force	Speed of the vehicle, Distance to next obstacle, Feedback from ABS system(corrections on-the-fly if needed) Increase sampling speed, etc.
Automatic deceleration (no brake, engine deceleration)	
Automatic deceleration(brakes, engine deceleration)	

Challenges in RL

- Training the agent:
  - Combination of (action, state):
    - Increases as the agent explores
  - Growth of the Q-table becomes “stable” after training
    - Related to the explore-exploit ratio
    - However, at this point, the Q-table will be extremely big
  - Training is complete: now save and deploy the model:
    - What are some problems?





Challenges in RL

- If using Q-learning:
  - For the model to make an effective response to avoid a collision:
    - Needs to look for the proper Q-value:
      - Needs to match sampled scenario with Q-value
  - The model saves parameters according to what it was configured to save:
    - Defined limits

Challenges in RL

- Some problems:
  - Action space and state space are large
  - The environment contains “other factors”
  - Regardless of the approach (SARSA, Q-Learning):
    - Problems with generalization
    - Generalization:
      - the ability of the same developed model to predict given multiple environments.
      - the ability of the same developed model to identify irregularities and make corrections “on the fly”
- Can excel in:
  - Industry Automation, why?
  - Trading and Finance, why?

Challenges in RL

- To remedy these challenges:
  - Deep Learning (DL)
    - Increases the processing scope
      - Instead of prediction, an approximation is calculated
    - Can handle the larger environment
      - “other factors”
  - Deep Reinforcement Learning (DRL):
    - More powerful than a DL approach by itself
    - Implements RL Q-learning to increase the processing scope

THE UNIVERSITY OF
SOUTHERN MISSISSIPPI®