# AIDL_B_02 Final Project

## "Siamese" Network

**Objective:** Its goal is to predict which action has been selected.

**Description:** This network consists of an ***Encoder*** (i.e., an CNN network, could be 2D, 3D, ConvLSTM, etc) and an ***Inverse Model*** (dense neural network). It takes as input 4 stacked frames $S_{t0}, S_{t-1}, S_{t-2}, S_{t-3}$ represented as $S_t$ (Figure 1) and the another 4 stacked frames $S_{t+1}, S_{t0}, S_{t-1}, S_{t-2}$ (sliding window with overlap 75%) represented as $S_{t+1}$ (Figure 1). After several convolutional/pooling layers the outputs (encoded $S_t$ and $S_{t+1}$) are flattened and concatenated (we mentioned 200d arrays in the course that could create a 400d array, **but the dimensions are up to you, but keep in mind to select the same for the action embeddings in page 3**). The concatenated array is passed to the ***Inverse Model*** to produce 5 probabilities. Using argmax you should select the predicted action and then compare it with the actual one $a_t$ **use categorical cross entropy or negative log likelihood as a loss function (keep in mind they are not the same!)**.

**Exercise:** You should develop a **siamese.py** file. Inside there should be:

- a class called encoder model, taking as input the stacked frames and producing the extracted features (flattened array).
- a class called inverse model, taking as input two flattened arrays, concatenating them and passing them to fully connected layers.
- a function called loss function, taking as input the output actions (i.e., 5), and the target action, outputting the loss. The output actions will be passed to argmax or sth similar to get the predicted action, and the selected loss function can be hardcoded.

**More details:**

- the network hyperparameters is up to you and it ok to be completely arbitrarily chosen
- be aware that in pytorch NLLLoss and CrossEntropyLoss are not the same, ne careful which one to choose.
- you could use one class for both the encoder and the inverse model (e.g., siamese model).
- you can have a look here for tips.

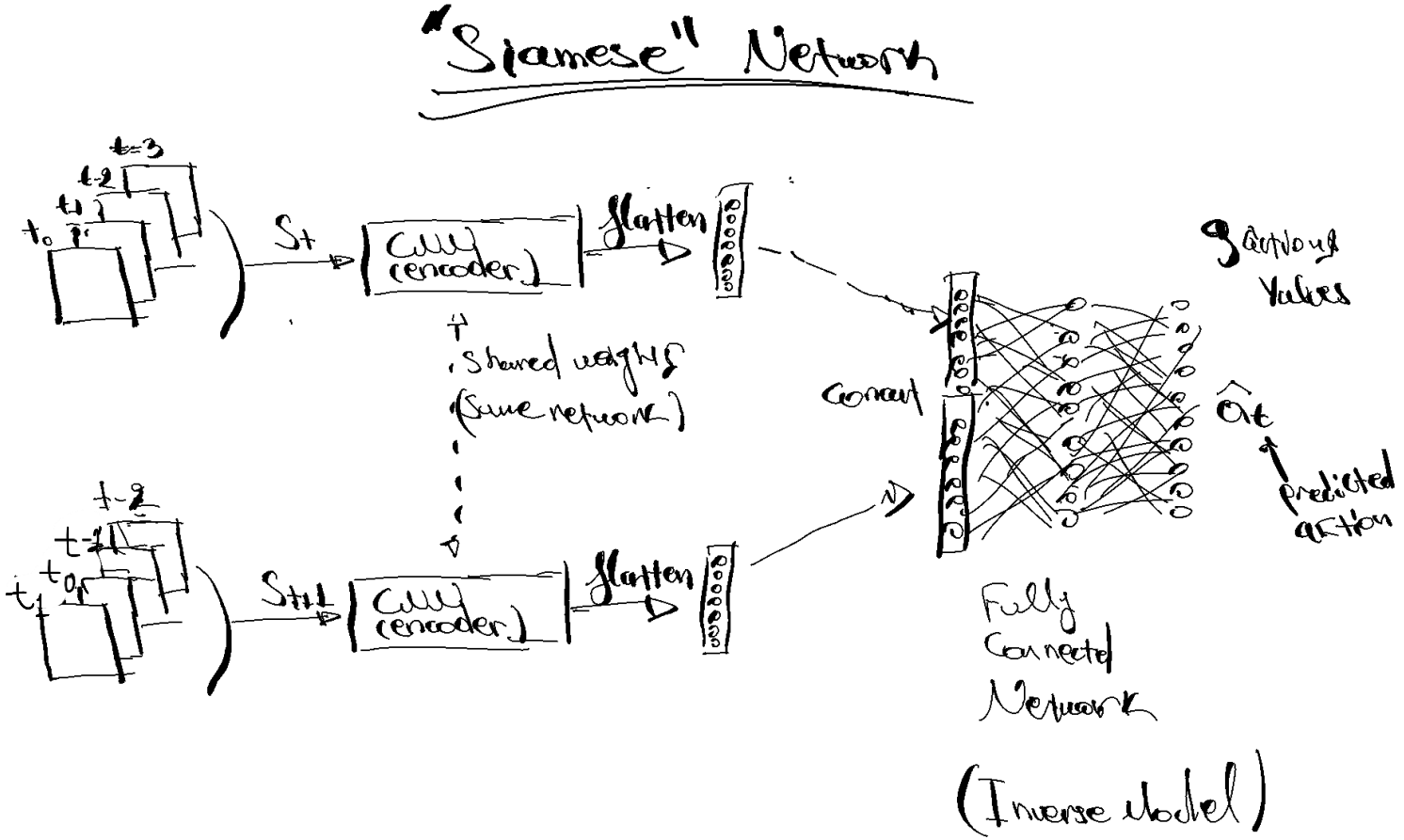



*Figure 1. Abstract representation of the "Siamese" Network.*

# Action Embeddings Network

**Objective:** Its goal is to predict the encoded next state.

**Description:** This network takes consists of the ***Encoder*** (**should be exact same architecture used for the Siamese part loaded and the parameters should be trainable!!!**), an ***Embedding*** network and a ***Forward Model*** (dense neural network). The ***Encoder*** takes as input 4 stacked frames $S_{t0}, S_{t-1}, S_{t-2}, S_{t-3}$ represented as $S_t$ (Figure 2). As mentioned, it is the same network you developed before. The ***Embedding*** network takes as input the $a_t$ and passes it to an embedding layer (**select same embedding size with the states**). The ***Forward Model*** takes as input the action embeddings and the encoded state, concatenates them and passes them to dense layers. The output shape should be the same with the encoded $S_{t+1}$(4 stacked frames $S_{t+1}, S_{t0}, S_{t-1}, S_{t-2}$), for example 200d array. Using **root mean squared error as loss compute the error** between the predicted encoded $S_{t+1}$ and the actual one.

**Exercise:** You should develop an **action_embeddings.py** file. Inside there should be:

- a class called encoder model, taking as input the path of a state encoder, and the stacked frames to produce the extracted features (flattened array). It should use the same existing model as before!
- a class called embedding model, taking as input an action, creating the one-hot encoded representation, pass it to an embedding layer.
- a class called forward model, taking as input the embedded action and the encoded state, concatenate them and pass them to dense layers producing the predicted encoded next staked frames.
- a function called loss function, taking as input the predicted encoded $S_{t+1}$ and the actual one, outputting the RMSE loss. The loss function can be hardcoded.
- a function called store embeddings, that will store the actions embeddings to be used for the next exercise.

**More details:**

- the network hyperparameters is up to you and it ok to be completely arbitrarily chosen, due to computational resources needed
- you can have a look here.

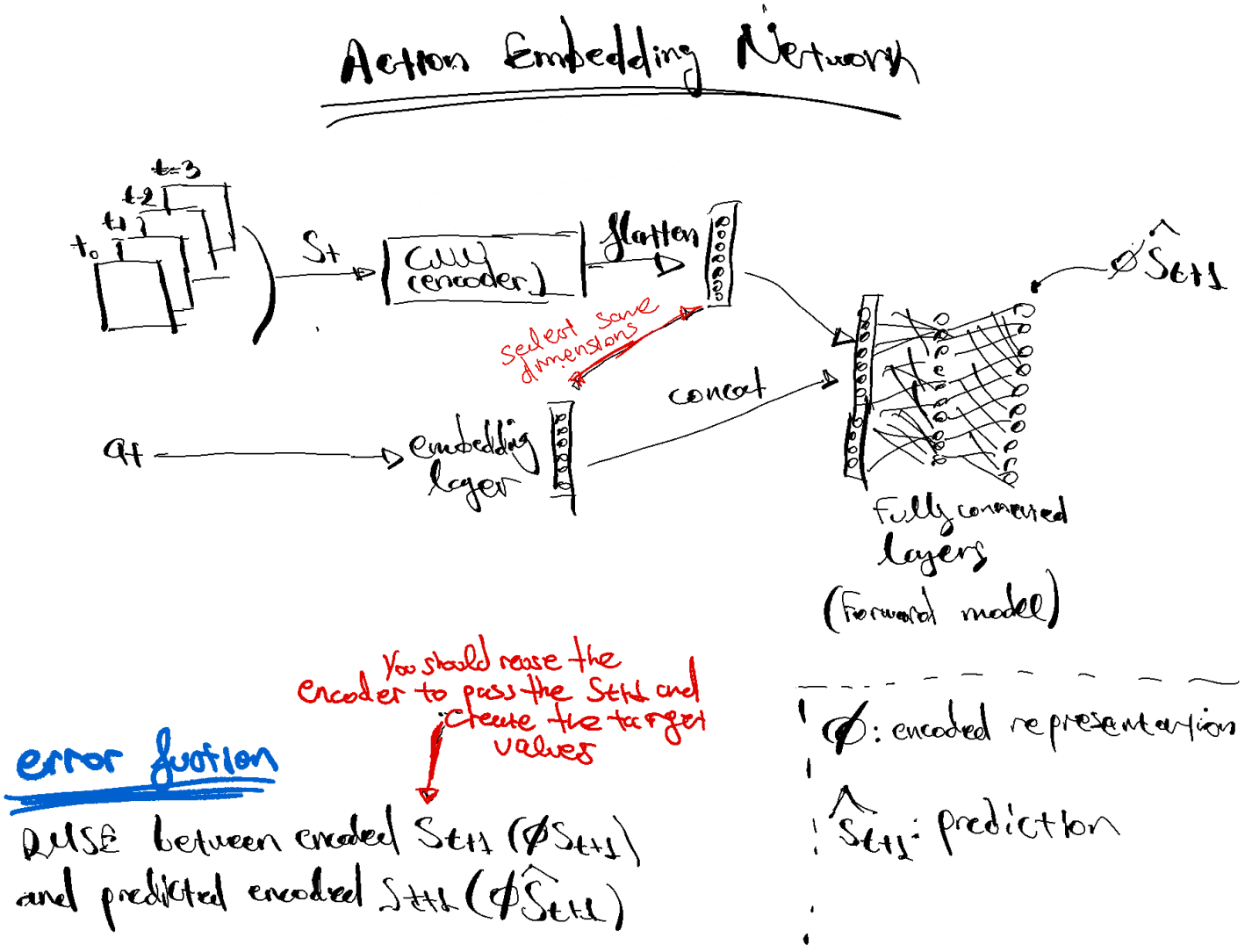



*Figure 2. Abstract representation of the Action Embeddings Network*

## Train the Networks (Siamese, Action Embeddings)

**Objective:** Its goal is to train the two networks simultaneously.

**Description:** Check Figure 3.

**Exercise:** You should develop a **icm.ipynb** file. Inside there should be:

- the two networks defined earlier
- the loader for the dataset you collected (split them in train, test, valid this is not RL)
- the training process, **use 1 total loss.** It could total_loss=siamese_loss+action_loss, or total_loss=b*siamese_loss+(1-b)*action_loss, beta is a hyperparameter
- the results (metrics for the two networks)
- save the action embeddings and the state encoder.

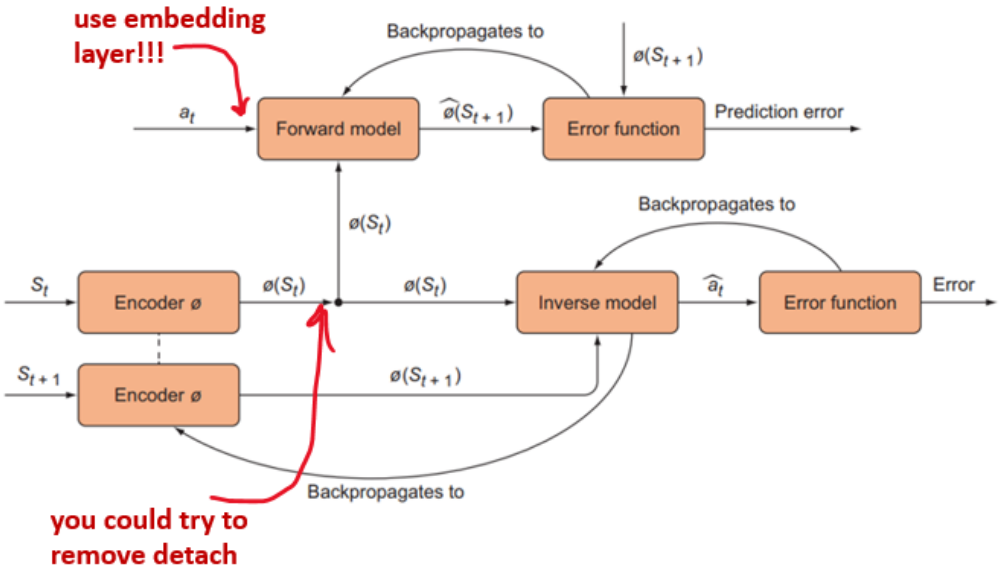



*Figure 3. Abstract representation of the "ICM" Network.*

# Model-based DQN on “AlienDeterministic-v4"

**Objective:** Its goal is to combine action embeddings and the stacked input frames to train a better DQN.

**Description:** This network consists of the ***Encoder*** (**should be exact same architecture used for the Siamese part loaded and the parameters to be finetuned**), **all the action *Embeddings*** (should the same used before), a **Multi-head cross attention (MHCA) layer** and a ***FC Model*** (dense neural network). The ***Encoder*** takes as input 4 stacked frames $S_{t0}, S_{t-1}, S_{t-2}, S_{t-3}$ represented as $S_t$ (Figure 2). As mentioned, it is the same network you developed before. The ***MHCA*** takes as input all the action embeddings and the encoded state, fuses them and passes them to dense layers (**FC**). The output shape is the 6 action-values (think of a way to include ‘Fire’ and keep in mind the difference in the indices).

**Exercise:** You should develop a **mb_dqn.py** file. Inside there should be:

- a class called encoder model, taking as input the path of a stored encoder (let’s assume that’s a trained one), and the stacked frames to produce the extracted features (flattened array). It should load the existing model.
- a function loading the stored action embeddings.
- a class called MHCA model, taking as input the embedded actions and the encoded state, pass them through a multihead attention layer, (Q=encoded_state, K=action_emb, V=action_emb), then a layer normalization, flatten them and pass them to dense layers producing the predicted q-values.

**More details:**

- the network hyperparameters is up to you and it ok to be completely arbitrarily chosen
- the encoder and embeddings should be finetuned
- you can have a look here, and here.
- Use Q-loss function

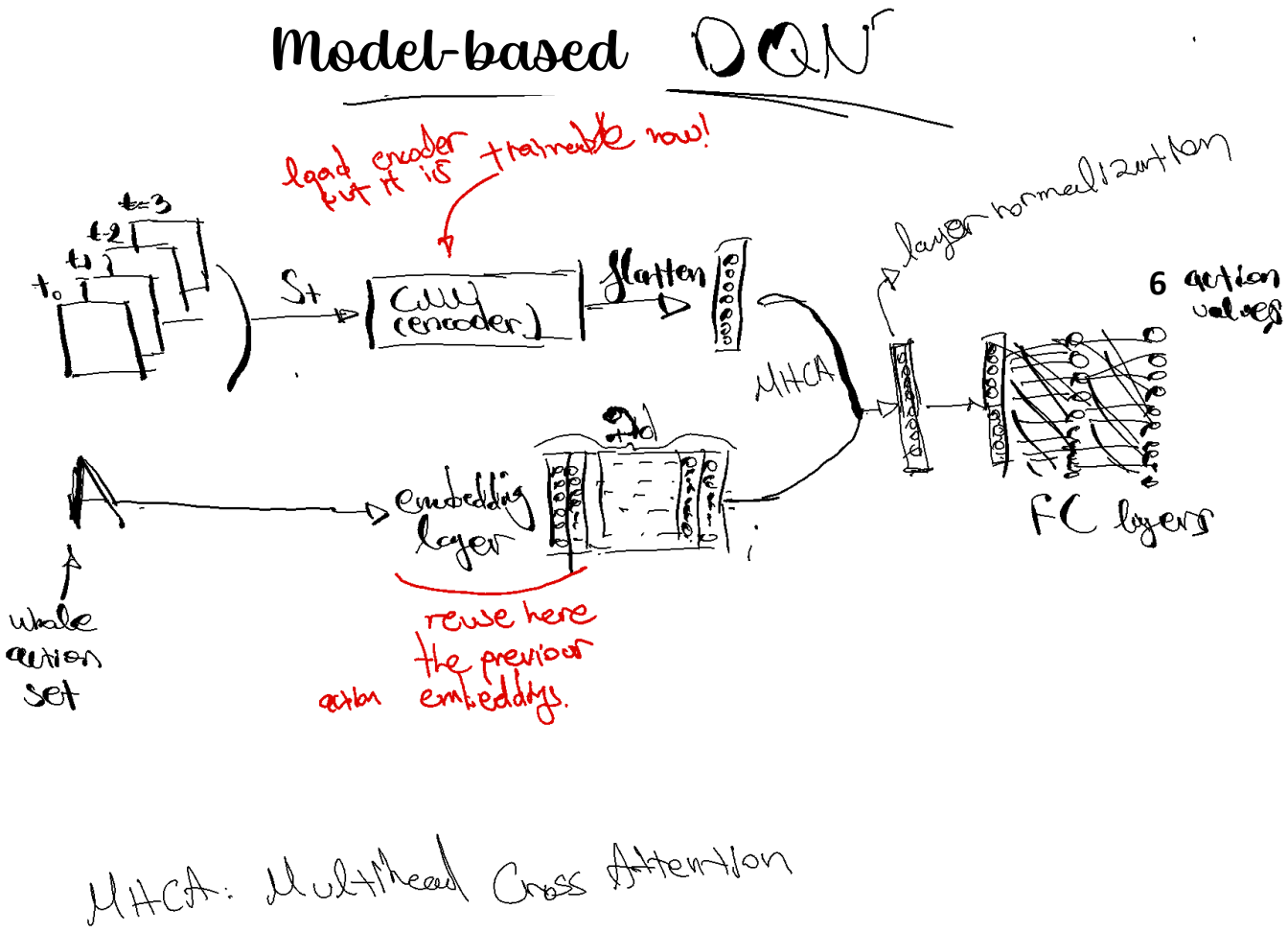



*Figure 4. Abstract representation of the Model-based DQN*

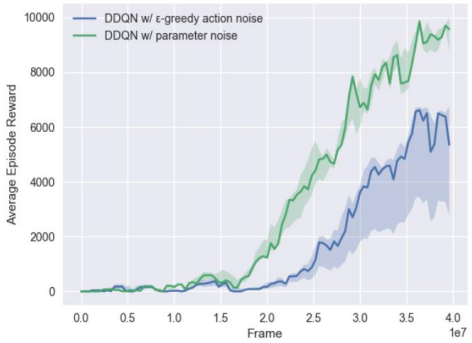
# DQN on “FreewayDeterministic-v4"

**Objective:** Its goal is to predict the action values.

**Description:** Develop the whole training pipeline for DQN. The **agent** (i.e., model) takes as input 4 stacked frames $S_{t0}, S_{t-1}, S_{t-2}, S_{t-3}$. After several convolutional/pooling/dense layers it outputs the predicted action values. Create a **replay buffer**, **target network**, **loss function** etc. to train the agent appropriately.

**Exercise:** You should develop a **DQN.ipynb** file. Inside there should be:

- Use **"FreewayDeterministic-v4"**
- All the necessary functions and the results plotted nicely (present your NN architectures and the obtained results using average episode rewards)




- Create a .ppt presentation for this and could also record a trained agent.

**More details:**

- the network hyperparameters is up to you but it is **NOT ok to be completely arbitrarily chosen**
- you can have a look here for frame stacking.
- DQN training, replay buffer, target network, are presented here
- The Atari environment installation, API for PONG is here, the Freeway game is exactly the same, having less actions