Albert-Ludwigs Universität Freiburg
Institut für Informatik
Deep Learning Lab **Excercise Sheet 3**
José Arce y de la Borbolla 4.12.2018

Deep Learning Lab
Excercise Sheet 3

4.12.2018

Abstract

The purpose of this report is to present the reasoning behind the design choices for solving the OpenAI Gym Car Racing Environment using a supervised machine learning model through a combination of 2D convolutions and Long Short-Term Memory Cells using the TensorFlow framework, and to show the results obtained with this approach. For instructions on testing the agents, jump to Section 3.4.

1 Environment Definition

The environment in which the problem takes place consists of a random generated track circuit with corners of varying difficulty.

The objective is to drive a racing car around the track, visiting every section of the track in the least possible time to maximize the value of the reward. The observations of the environment include an eagle view of the portion of the track the car is currently at, the accumulated reward value and a couple of useful indicators such as the overall speed or the position of the steering wheel, as can be seen in Figure 1. The reward function is calculated as follows: for every passing step, 0.1 is subtracted from the reward, for every newly visited tile $1000/N$ is added to the score, where N is the total number of tiles in the track. The episode ends when all the track tiles have been visited, or after a certain amount of simulation steps have taken place.

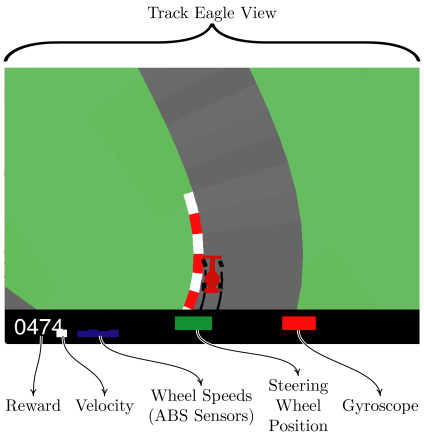
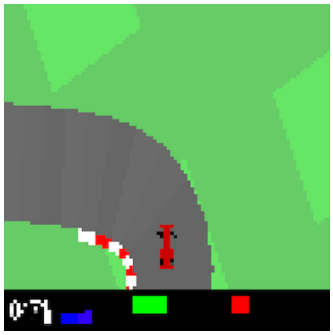


Figure 1: Environment Definition.

2 Data.

2.1 Data Collection

Gathering data is pretty simple: the user plays the game manually while a process takes snapshots of the environment and records the user actions for each simulation step. The snapshots of the



(a) Sampled Observation.

User Action	Key	Stored Vector
Go Straight		$[0 \quad 0 \quad 0]$
Turn Left	$[\leftarrow]$	$[-1 \quad 0 \quad 0]$
Turn Right	$[\rightarrow]$	$[1 \quad 0 \quad 0]$
Accelerate	$[\uparrow]$	$[0 \quad 1 \quad 0]$
Break	$[\downarrow]$	$[0 \quad 0 \quad 0.2]$

(b) Sampled User Actions.

Figure 2: Sampled Data.

environment are downsampled 96×96 pixel (originally RGB) images whereas the user actions were originally encoded as a vector with three components, allowing superposition of actions, as shown in Figure 2.

The original code was modified to allow the user to play in multiple sessions without rewriting the previous session's data, but rather to append the new data to the previously stored one. This allows a better collection of data, as it is easier to collect more information, and in a more varied way due to changes in playing style over different days.

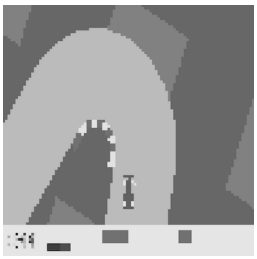
Though the original intention was to build a dataset with at least 100'000 samples, it soon became obvious that, due to memory limitations in the computers, it would be impossible to load the entire dataset into RAM and do any kind of preprocessing in the entirety of the data. For that reason, the dataset was originally comprised of two larger data files with 50'000 and 40'000 samples, and a couple with only 5'000 samples. After rewriting the action_to_id function, a new dataset was created with 60'000 samples. In order to save precious time and resources, because of the hardware limitations, preprocessing of the data was also shifted to the collection pipeline, instead of the training one.

In a futile attempt to also save storage space, the rewards and terminals were dropped from the saving process, while the images were stored as 8 bit gray-scales and the actions as one hot vectors. No matter how the data was stored, the resulting files were always larger than if no preprocessing had been done, most likely due to making the gzip compression process less efficient. However, loading times for training were significantly improved.

2.2 Preprocessing

As stated in the previous section, the preprocessing pipeline was shifted to the collection one to improve performance, as preprocessing took place on batches of only 1'000 samples at a time, instead of the whole 50 or 60'000 samples of a complete dataset, which proved to be undoable in the pool machines due to crashing because of memory limitations.

Because the color information could be thought of as redundant or not providing more insight to the state of the environment, image data was converted to 8 bit grayscale to save training and processing power. Due to the selection of Softmax as the activation function for the last output layer, it restricted the model to taking a single discrete action at each step. For that reason, the user actions were encoded as one-hot vectors of length 5 and stored as such in the dataset. An example of the new preprocessed data can be seen in Figure 3.



(a) Preprocessed Observation.

User Action	Key	Stored Vector
Go Straight		[1 0 0 0 0]
Turn Left	[←]	[0 1 0 0 0]
Turn Right	[→]	[0 0 1 0 0]
Accelerate	[↑]	[0 0 0 1 0]
Break	[↓]	[0 0 0 0 1]

(b) Preprocessed User Actions.

Figure 3: Preprocessed Data.

2.3 Resampling

Due to the nature of the game, most of the time the player is not performing any actions, but rather just going straight. Also because the objective is to go around a circuit in a counter-clockwise fashion, most of the turning takes place to the left. Given the greedy nature of the player, braking is seldomly used. At the same time, a user might perform different actions to seemingly equal looking situations, which makes a model have a hard time learning highly non-deterministic labeling of samples. The distribution of actions for all the recorded data samples up to this point (around 90'000 actions) can be seen in Figure 4.

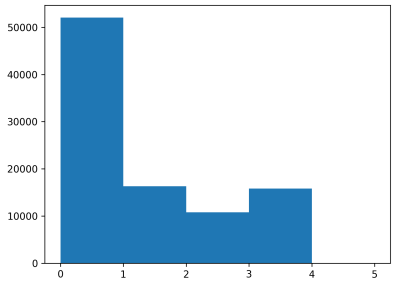


Figure 4: Original Action Histogram.

It is worth noting that the braking action (5) never showed up in the original dataset. This is partly due to the greediness of the player, but also because it was normally used in combination of another action, usually when turning sharply around u-turns. The `action_to_id` function only cares about single actions at any given step, and if one were to take more than one, then it would encode it as going straight. After (rather later than sooner) realizing this, `action_to_id` was rewritten to take into account multiple actions, in which case it randomly picks one of them so as to not loose so much data in the process. This allowed the distribution of the actions to look a bit more natural, but still biased towards inaction, then with almost equal proportions to accelerating and turning left, then to turning right, and finally to braking. The histogram of the data sampled with the new function can be seen in Figure 5.

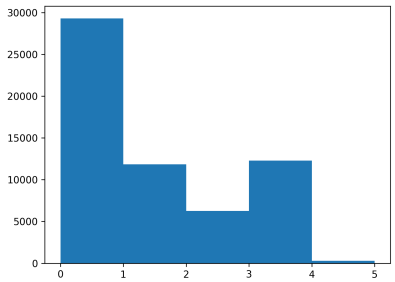


Figure 5: Action Histogram with new `action_to_id` function.

Because of the reasons stated in the previous paragraph, a resampling function was im-

plemented before providing the model the data for training. It's purpose: try to balance the distribution of labels more uniformly across the training data.

The resampling function does away with a lot of data, by assigning a probability depending on an action's occurrence in the dataset. Thus, the more an action shows up in the original dataset, the less likely is it going to appear in the resampled one. Given that resampling deletes frames randomly according to a probability distribution to create a sampled dataset that is some percentage smaller than the original one, it was only used to train purely convolutional models, because doing so for models involving recurrence would mean breaking perfectly reasonable sequences of data into nonsensical chunks of images and actions which might not even be possible to happen in a real simulation. Figure 6 shows an example of the distribution histogram of the 60k dataset after being resampled to 75%. Given that the resampling operation is random, every training session would get a different distribution.

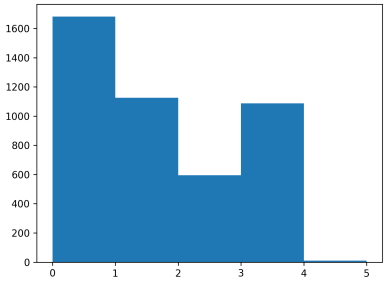


Figure 6: Action Histogram after resampling to 75% of the data.

In order to select whether sampling is desired from the command line without having to alter the code, the boolean flag `--resample` was added to the train_model.py program. An arbitrary value of 75% was chosen for the resampling rate, meaning that only 75% of the original data will remain after randomly selecting samples according to a probability inversely proportional to its occurrence in the original dataset.

Due to the limitations of memory, every single action had to be done in batches. Training was of course accomplished doing so, but also evaluation, preprocessing, resampling and re-sequencing (more on that later).

2.4 Data Augmentation

The easiest way to augment the data would have been to mirror the images along the vertical axis and, if a turning (either left or right) action was taken, mirror the action too. The problem with this approach would be the mirroring of the bottom indications, which in case the network used them to derive a feature, would then be totally useless. If they were useless in the first place, then it would even make sense to crop the images and retain only the eagle view to make processing faster, and then do the mirroring to augment the data. If the information were useful, then the mirroring of the eagle view could take place, but then the steering wheel, ABS sensors and gyroscope indicators would need to be mirrored in place. The other issue issue is that it only helps in bridging the gap between the turning operations, doing nothing for the other actions.

The second reason, and perhaps the most important one, would be the placing of that data within the data set. Given that the sequencing operation groups consecutive images into fixed length time steps for the training of the recurrent neural network to happen, then the question remains of where to insert the new images, such that they make sense when put together in a time sequence.

Although technically feasible, it was decided against data augmentation for the training of all the models for the reasons stated in this section.

3 Model Design

Thinking ahead, the design process of the model contemplated the recurrent part of the network since the beginning, though always giving the possibility of not using it by leaving the ''`lstm_layers`'' key of the configuration dictionary as an empty list, as is the case for models **net1** and **net2**. The general architecture for all the models used was a succession of several convolutional layers, followed by the recurrent portion of the network comprised of LSTM cells and finally a couple of fully connected layers. Models following this general architecture can be created at will by simply defining their properties in a json file with extension `.narq.json`. The purpose of this is to be able to reuse different models for training and testing, without having to change any single bit of code. Used in conjunction with TensorFlow's own checkpoint files made it a breeze to change models on the fly. The actual configuration files for the models used can be found in Appendix A.

3.1 CNN + RNN

The way the network was designed completely contradicts the intended way of integrating the time dimension into the channel dimension. Not only would that defeat its purpose of handling multi-channel images such as pictures with color information, but in turn would also be completely lacking of any significance. After the first convolution, the convolved images across the different time steps would be added together, loosing any relevant temporal information that the recurrent network could use. Moreover, the "temporal" dimension size would change between convolutional layers, as the varying number of filters dictate.

For that reason, it was best though to incorporate a 5^{th} dimension to the input tensor, such that it's shape would look like:

$$\left[\begin{array}{ccccc} B & T & W & H & C \end{array}\right]$$

where:

B is the Batch dimension, or the number of samples. Of dynamic length and mostly used for training.

T is the Temporal dimension, or the number of time steps in the sequence. Must be defined during the graph creation and kept constant throughout all training and testing sessions. This affects the convolutional layers, and of course the static LSTM cells.

W is the Image Width in pixels.

H is the Image Height in pixels.

C are the Image Channels. E.g.: 3 for RGB color images or 4 for RGBA for color images with transparency. In this particular example it is of size 1 as the images are in Grayscale.

The way the convolutional layers handle the 5D input is, simply put, that they don't. Unless one uses 3D convolutions, which for this particular problem makes no sense, convolutional layers in TensorFlow don't like 5D inputs.

The idea behind the design follows that of the nature of recurrent neural networks themselves. Just as they are "unfolded" or "unrolled" to calculate their internal and cell states while sharing weights for each of the inputs in a sequence, the convolutional layers also unfold, sharing the same kernels and layer properties among all inputs. The architecture can be though of as running

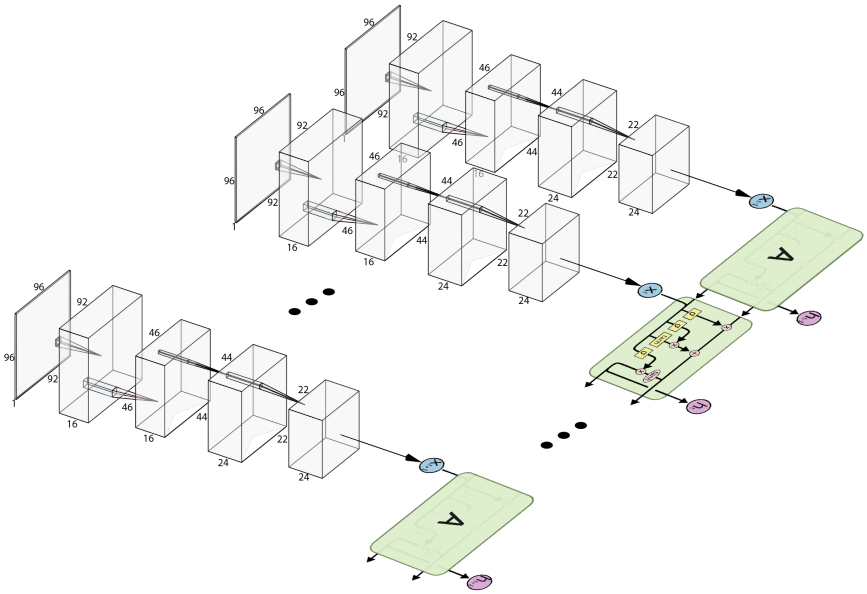


Figure 7: CNN + RNN Architecture.

the same convolutions for each of the inputs in parallel, thus constructing the same feature maps for each one of the input images in a sequence.

As there is no way of dynamically doing this in TensorFlow at the moment, the Temporal dimension must be fixed starting from the design phase, just as any other hyper-parameter. Once a value has been selected, the model will create T convolutional layers for each of the originally (and properly called layers), with the property that they will all share the same filters at each level. Conceptually, this could be visualized as shown in Figure 7. The resulting network can be seen in the TensorBoard Graphs tab and Appendix A, confirming that the desired design was successfully implemented.

Before the first convolutional layer, the tensor is unstacked along it's temporal dimension, and then each individual image is fed to it's own convolutional neural network, only for the resulting feature maps to be flattened and stacked again in preparation for the recurrent part of the network.

The great advantage of this approach is that the number of parameters does not change with respect to using a single convolutional network, while at the same time we preserve the temporal dimension untouched until it reaches the LSTM cells where it matters the most. The clear disadvantage: a huge increase in training time due to the gradients coming from the recurrent network need to be multiplied with the convolution of each image in the sequence.

3.2 Re-sequencing.

After the data set has been split into training and validation data, the selection of samples for the batches used for training was done by randomly generating a list of indices, so as to "shuffle" the data, while at the same time not having to actually shuffle all the data at once, as that would be more expensive. Given that the network design is inherently recursive, after randomly selecting indices for the batch, the states preceding the selected indices need to be used, so that

the input to the LSTM makes actual sense.

For example, for images $I_0, I_1, \dots$ up to I_n, their corresponding sequence for a history length of $h = 4$, along with the corresponding action labels a_i would look like the following representation., where the sliding sequence window is represented by the square brackets.

$$
\begin{array}{cccccccc}
 & & & a_3 & & & & \\
 & & & \downarrow & a_4 & & & \\
[\mathbf{I_0} & \mathbf{I_1} & \mathbf{I_2} & \mathbf{I_3}] & \downarrow & a_5 & & \\
I_0 & [\mathbf{I_1} & \mathbf{I_2} & \mathbf{I_3} & \mathbf{I_4}] & \downarrow & a_6 & \\
I_0 & I_1 & [\mathbf{I_2} & \mathbf{I_3} & \mathbf{I_4} & \mathbf{I_5}] & \downarrow & \dots \\
I_0 & I_1 & I_2 & [\mathbf{I_3} & \mathbf{I_4} & \mathbf{I_5} & \mathbf{I_6}] & \\
I_0 & I_1 & I_2 & I_3 & [\mathbf{I_4} & \mathbf{I_5} & \mathbf{I_6} & \dots
\end{array}
$$

From the example above can be noticed that the labels selected correspond to the action taken by the player at the last step of a sequence. Thus, the first $h - 1$ actions are not used at all. It was decided to not pad the sequence with black images, because loosing 5 or 10 actions, out of a dataset of 60'000 would not make much of a difference. On the other hand, while testing the agent, as the sequence queue is filling up, the program is designed to accelerate for the first $h - 1$ steps, after which the agent takes control with its predictions.

3.3 Training

Quite a few training sessions were performed on around 10 different models in total, eventually settling with the 5 most promising ones. During all these sessions, the model was never capable of exceeding 75% validation accuracy, despite having played with all the hyper-parameters in every possible way: extending the number of batches to train on, changing the datasets, early stopping and saving the model at certain checkpoints, retraining a model saved to a checkpoint even with different data, changing the learning rate, reducing the number of parameters so as to better generalize, adding and removing layers, adding drop-out to the recurrent and fully connected layers and L2-regularization to the loss function, and many other actions. Regardless of this shortcomings, the model ended up performing quite good in the test track, perhaps due to the limited amount of unique states that can exist in the simulation.

The learning curves for the 5 most promising agents can be seen in Appendix B, as well as in the interface of Tensorboard for greater detail.

3.3.1 Optimization Algorithm

For training the model, an ADAM optimizer was used. Originally it considered only the cross entropy loss function over the labels and predicted outputs, but after not being able to get past a 75% validation accuracy, L2-regularization was also applied only to the weights/kernels of all the layers.

3.4 Testing

To be able to test the agent, the data obtained from the simulator must be preprocessed int the exact same fashion as it was done for the training. Similarly, the outputs predicted by the model must be converted from their one-hot format to the vector with 3 components that the environment takes.

As stated in Section 3.2, because of using recurrent neural networks, the first $h - 1$ actions taken by the agent are fixed to accelerate, where h is the history length of the sequence input

Model	Training				Testing	
	Final Training Accuracy	Final Validation Accuracy	Number of Batches	Training Time	Reward	
					Mean	σ
net1 (CNN)	86.6481%	72.3833%	100'000	40:22	585.1061	307.7683
net2 (CNN)	72.7296%	62.1167%	300'000	1:06:22	809.8464	111.2685
net3 (CNN+RNN)	91.9944%	71.7333%	200'000	11:00:00	566.9941	300.5476
net4 (CNN+RNN)	92.6074%	72.4333%	200'000	3:45:02	748.0682	160.8649
JABnet (CNN+RNN)	94.9185%	72.8333%	200'000	6:01:08	742.1062	221.1171
Myself (Human NN)	-	-	-	~1:00:00	884.3021	41.3228

Table 1: Exercise Results.

to the agent. This only applies for recurrent agents while the step queue is being filled, as the history length for convolutional agents must be equal to 1.

In order to test the agent, it is necessary to load both the json architecture file and the desired tensorflow checkpoint file. This can be done by using the following added arguments when calling the program from a command line.

```
python3.5 [-u] test_agent.py --agent <agent_name> [--debug <debug_level>]
```

where:

agent_name the name of the model to be tested. It can currently take one the following values: `net1|net2|net3|net4|JABnet`.

debug_level a variable to control the amount of output of the program. For this script, any value above 1 will result in outputting the agent's actions to stdout.

If more control over the actual files is required, the following parameters can be used:

```
python3.5 [-u] test_agent.py --arq_file <path_to_narq> --ckpt <path_to_ckpt>
[--debug <debug_level>]
```

where:

path_to_narq is the full or relative path to the network architecture file with extension `.narq.json`. Located in the `./models/` subdirectory in this project.

path_to_ckpt is the full or relative path to the TensorFlow checkpoint containing the saved configuration and parameters for the model. Also located in the `./models/` subdirectory.

4 Results

The following table summarizes the results obtained by each model during training and testing. Only the values for the last training and testing sessions are shown.

5 Discussion

After having programmed an agent with two different approaches, one using only convolutional layers and one using both convolutional and recurrent layers, and having seen the results, it might seem obvious that the purely convolutional one should suffice for solving this task. However,

perhaps due to bias while becoming fond of recurrent networks, I believe that the model using LSTM cells performed more cleanly, with a lot less jerking thanks to it's capacity to see the corners 5 or 10 steps ahead.

During this exercise, the quality of the data used for training became very apparent, affecting the entire distribution of the actions predicted by the model. The same applies to any process directly connected to the data, such as shuffling, selecting, preprocessing and even storing.

Although the behavior observed during the training sessions clearly screams of overfitting the training data, it seems that it did not affect the performance in the test environment so much. It is my hypothesis that the model was never able to overcome the 75% validation accuracy barrier, despite having used all the tricks in the book, due to the extremely indeterministic process of labeling the data. I strongly believe that there must likely be sets of equally looking images with totally contradicting actions taken during the collection drives. The hypothesis is, as of this moment, lacking any evidence due to time constraints, but eventually I shall have a look on the entire dataset to confirm it. The moral of the story: That is why one should not drive after having had a couple of drinks.

Appendices

A Appendix A: Network Architectures

During the course of the exercise, several architectures were tested in a search for the best agent. This appendix shows the json configuration files used to create the networks and the model graphs as generated by tensorboard of the most successful model architectures. The first models, i.e. **net1** and **net2**, are purely convolutional and fully connected models. The rest have a combination of convolutional, recursive and fully connected layers as explained in Section 3.1. The parallel application of convolutions can be best appreciated in the **net4** model, where the node for the third convolutional layer has been expanded.

The train and save nodes have been cropped out of the images for convenience

A.1 net1

```
{
  "name": "net1",

  "path": "./models/",

  "in_image_width"  : 96,
  "in_image_height" : 96,
  "in_channels"     :  1,
  "history_length"  :  1,

  "out_classes"     :  5,

  "learning_rate"   : 0.0001,
  "dropout_rate"    : 0.8,
  "l2_penalty"      : 0.01,

  "conv_layers": [
    {
      "name"         : "conv1",

      "kernel size"  :  5,
      "filters"      : 16,
      "activation"   : "relu",
      "padding"      : "VALID" ,
      "stride"       : [1, 1, 1, 1],

      "pooling"      : "max",
      "pool ksize"   : [1, 2, 2, 1],
      "pool stride"  : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"         : "conv2",

```

```
      "kernel size" : 3,
      "filters"     : 24,
      "activation"  : "relu",
      "padding"     : "VALID",
      "stride"      : [1,1,1,1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    }
  ],


  "lstm_layers": [
  ],

  "fc_layers": [
    {
      "name"        : "fc1",
      "units"       : 100,
      "activation"  : "relu"
    },
    {
      "name"        : "fc2",
      "units"       : 30,
      "activation"  : "relu"
    }
  ]
}
```

Listing 1: models/net1_CNN.narq.json Configuration File.

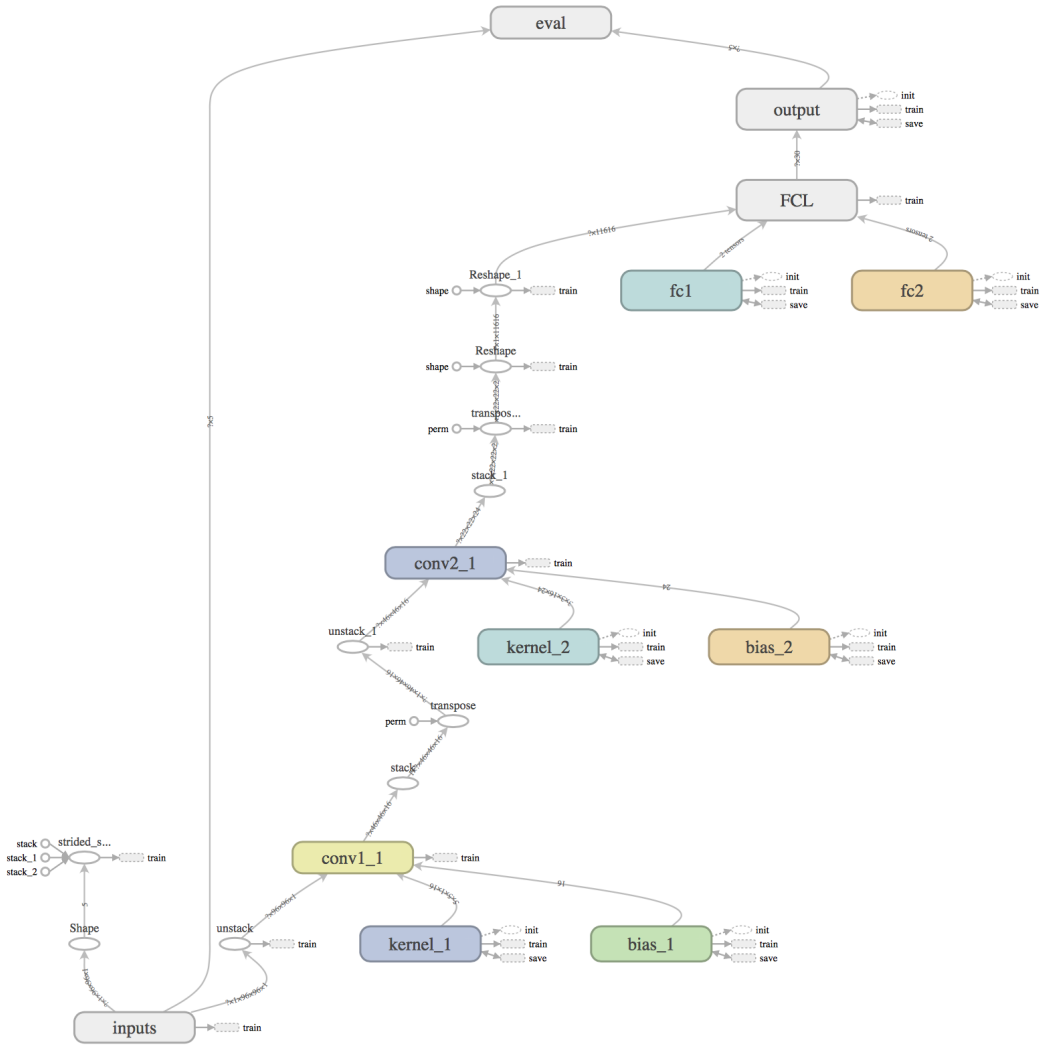


Figure 8: net1 FlowGraph.

A.2 net2

```
{
  "name": "net2_CNN",
  "path": "./models/",

  "in_image_width"  : 96,
  "in_image_height" : 96,
  "in_channels"     :  1,
  "history_length"  :  1,

```

```
  "out_classes"     :  5,

  "learning_rate"   : 0.00001,
  "dropout_rate"    : 0.8,
  "l2_penalty"      : 0.01,


  "conv_layers": [
    {
      "name"          : "conv1",

      "kernel size"   : 7,
      "filters"       : 3,
      "activation"    : "relu",
      "padding"       : "VALID",
      "stride"        : [1, 1, 1, 1],

      "pooling"       : "max",
      "pool ksize"    : [1, 2, 2, 1],
      "pool stride"   : [1, 2, 2, 1],
      "pool padding"  : "VALID"
    },
    {
      "name"          : "conv2",

      "kernel size"   : 5,
      "filters"       : 5,
      "activation"    : "relu",
      "padding"       : "VALID",
      "stride"        : [1, 1, 1, 1],

      "pooling"       : "max",
      "pool ksize"    : [1, 2, 2, 1],
      "pool stride"   : [1, 2, 2, 1],
      "pool padding"  : "VALID"
    },
    {
      "name"          : "conv3",

      "kernel size"   : 3,
      "filters"       : 10,
      "activation"    : "relu",
      "padding"       : "VALID",
      "stride"        : [1, 1, 1, 1],

      "pooling"       : "max",
      "pool ksize"    : [1, 2, 2, 1],
      "pool stride"   : [1, 2, 2, 1],
      "pool padding"  : "VALID"
```

```
    },
    {
      "name"          : "conv4",

      "kernel size"   : 3,
      "filters"       : 10,
      "activation"    : "relu",
      "padding"       : "VALID",
      "stride"        : [1, 1, 1, 1],

      "pooling"       : "max",
      "pool ksize"    : [1, 2, 2, 1],
      "pool stride"   : [1, 2, 2, 1],
      "pool padding" : "VALID"
    }
  ],


  "lstm_layers": [
  ],


  "fc_layers": [
    {
      "name"          : "fc1",

      "units"         : 400,
      "activation"    : "relu"

    },
    {
      "name"          : "fc2",

      "units"         : 100,
      "activation"    : "relu"
    },
    {
      "name"          : "fc3",

      "units"         : 30,
      "activation"    : "relu"
    }
  ]
}
```

Listing 2: models/net2_CNN.narq.json Configuration File.

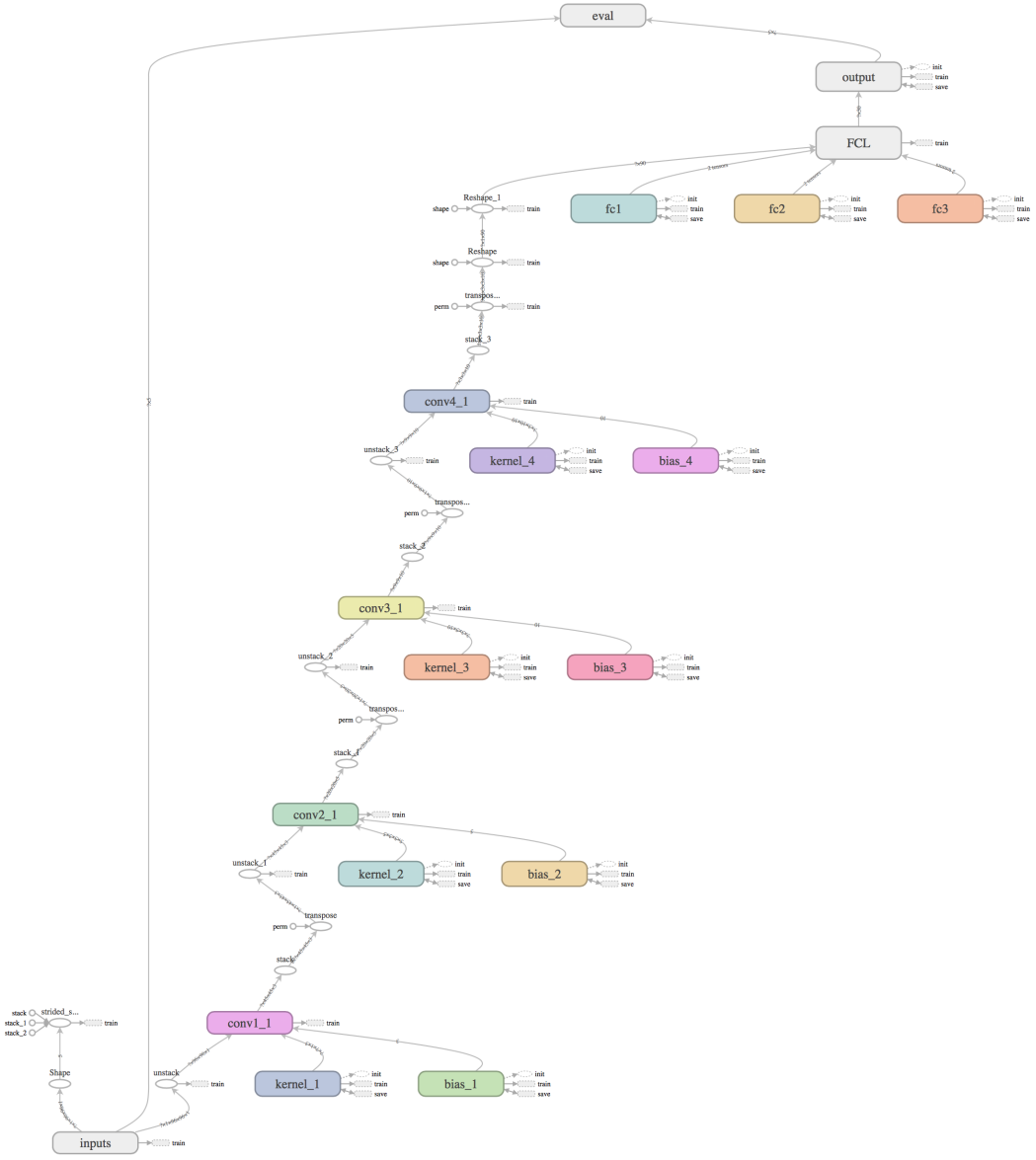


Figure 9: net2 FlowGraph.

A.3 net3

```
{
  "name": "net3_CRNN",
  "path": "./models/",

  "in_image_width"  : 96,
  "in_image_height" : 96,
```

```
  "in_channels"     :  1,
  "history_length"  : 10,

  "out_classes"     :  5,

  "learning_rate"   : 0.0001,
  "dropout_rate"    : 0.8,
  "l2_penalty"      : 0.01,


  "conv_layers": [
    {
      "name"        : "conv1",

      "kernel size" : 7,
      "filters"     : 3,
      "activation"  : "relu",
      "padding"     : "SAME",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"        : "conv2",

      "kernel size" : 5,
      "filters"     : 5,
      "activation"  : "relu",
      "padding"     : "SAME",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"        : "conv3",

      "kernel size" : 3,
      "filters"     : 10,
      "activation"  : "relu",
      "padding"     : "SAME",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
```

```
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"        : "conv4",

      "kernel size" : 3,
      "filters"     : 15,
      "activation"  : "relu",
      "padding"     : "SAME",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    }
  ],


  "lstm_layers": [
    {
      "name"        : "lstm1",

      "units"       : 200
    }
  ],


  "fc_layers": [
    {
      "name"        : "fc1",

      "units"       : 200,
      "activation"  : "relu"
    },
    {
      "name"        : "fc2",

      "units"       : 60,
      "activation"  : "relu"
    }
  ]
}
```

Listing 3: models/net3_CRNN.narq.json Configuration File.

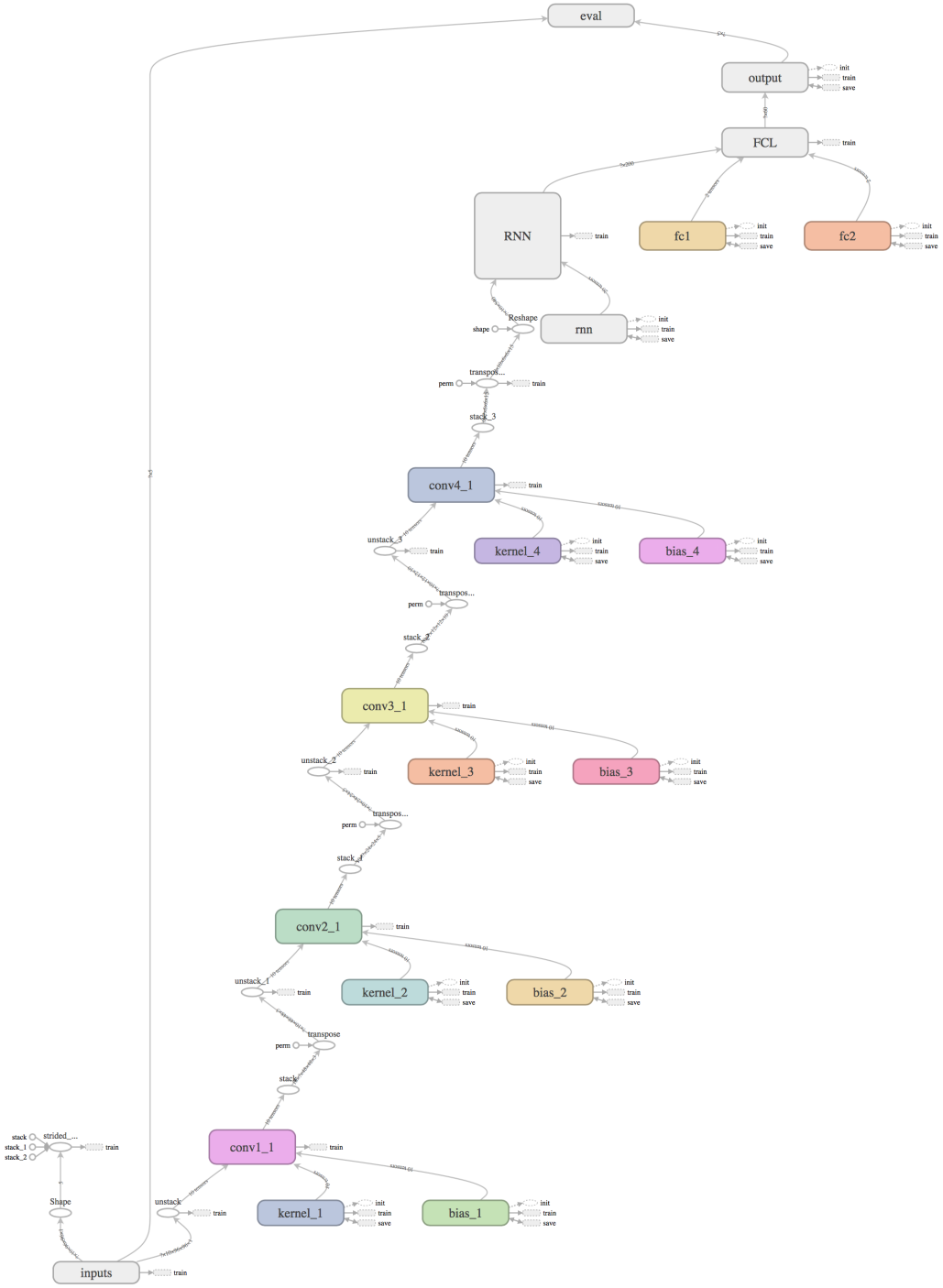


Figure 10: net3 FlowGraph.

A.4 net4

```
{
  "name": "net4_CRNN",
  "path": "./models/",

  "in_image_width"  : 96,
  "in_image_height" : 96,
  "in_channels"     : 1,
  "history_length"  : 5,

  "out_classes"     : 5,

  "learning_rate"   : 0.0001,
  "dropout_rate"    : 0.8,
  "l2_penalty"      : 0.01,

  "conv_layers": [
    {
      "name"        : "conv1",

      "kernel size" : 7,
      "filters"     : 3,
      "activation"  : "relu",
      "padding"     : "VALID",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"        : "conv2",

      "kernel size" : 5,
      "filters"     : 5,
      "activation"  : "relu",
      "padding"     : "VALID",
      "stride"      : [1, 1, 1, 1],

      "pooling"     : "max",
      "pool ksize"  : [1, 2, 2, 1],
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "VALID"
    },
    {
      "name"        : "conv3",

```

```
48        "kernel size" : 3,
49        "filters"     : 10,
50        "activation"  : "relu",
51        "padding"     : "VALID",
52        "stride"      : [1, 1, 1, 1],
53
54        "pooling"     : "max",
55        "pool ksize"  : [1, 2, 2, 1],
56        "pool stride" : [1, 2, 2, 1],
57        "pool padding": "VALID"
58      }
59    ],
60
61
62    "lstm_layers": [
63      {
64        "name"        : "lstm1",
65
66        "units"       : 20
67      }
68    ],
69
70
71    "fc_layers": [
72      {
73        "name"        : "fc1",
74
75        "units"       : 20,
76        "activation"  : "relu"
77      }
78    ]
79  }
```

Listing 4: models/net4_CRNN.narq.json Configuration File.

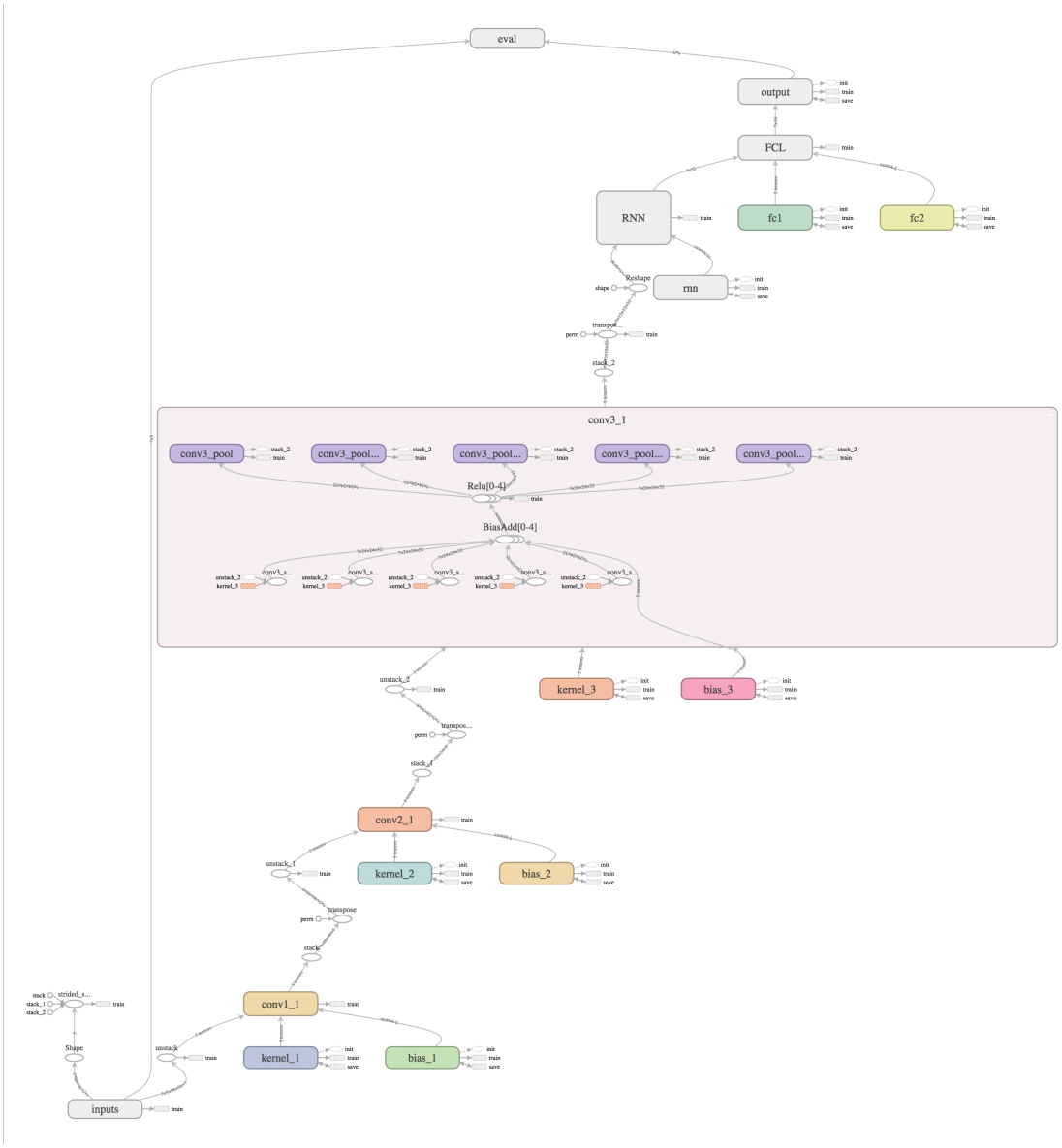


Figure 11: net4 FlowGraph.

A.5 JABnet

```
{
  "name": "JABnet",
  "path": "./models/",

  "in_image_width"  : 96,
  "in_image_height" : 96,
  "in_channels"     :  1,
```

```
  "history_length"   :  5,

  "out_classes"      : 5,

  "learning_rate"    : 0.0001,
  "dropout_rate"     : 0.8,
  "l2_penalty"       : 0.01,


  "conv_layers": [
    {
      "name"         : "conv1",

      "kernel size"  : 7,
      "filters"      : 4,
      "activation"   : "relu",
      "padding"      : "SAME",
      "stride"       : [1, 1, 1, 1],

      "pooling"      : "max",
      "pool ksize"   : [1, 2, 2, 1],
      "pool stride"  : [1, 2, 2, 1],
      "pool padding": "SAME"
    },
    {
      "name"         : "conv2",

      "kernel size"  : 5,
      "filters"      : 8,
      "activation"   : "relu",
      "padding"      : "SAME",
      "stride"       : [1, 1, 1, 1],

      "pooling"      : "max",
      "pool ksize"   : [1, 2, 2, 1],
      "pool stride"  : [1, 2, 2, 1],
      "pool padding": "SAME"
    },
    {
      "name"         : "conv3",

      "kernel size"  :  3,
      "filters"      : 32,
      "activation"   : "relu",
      "padding"      : "SAME",
      "stride"       : [1, 1, 1, 1],

      "pooling"      : "max",
      "pool ksize"   : [1, 2, 2, 1],
```

```
      "pool stride" : [1, 2, 2, 1],
      "pool padding": "SAME"
    }
  ],


  "lstm_layers": [
    {
      "name"         : "lstm1",

      "units"        : 32
    }
  ],


  "fc_layers": [
    {
      "name"         : "fc1",

      "units"        : 32,
      "activation"   : "relu"
    },
    {
      "name"         : "fc2",

      "units"        : 16,
      "activation"   : "relu"

    }
  ]
}
```

Listing 5: models/JABnet.narq.json Configuration File.

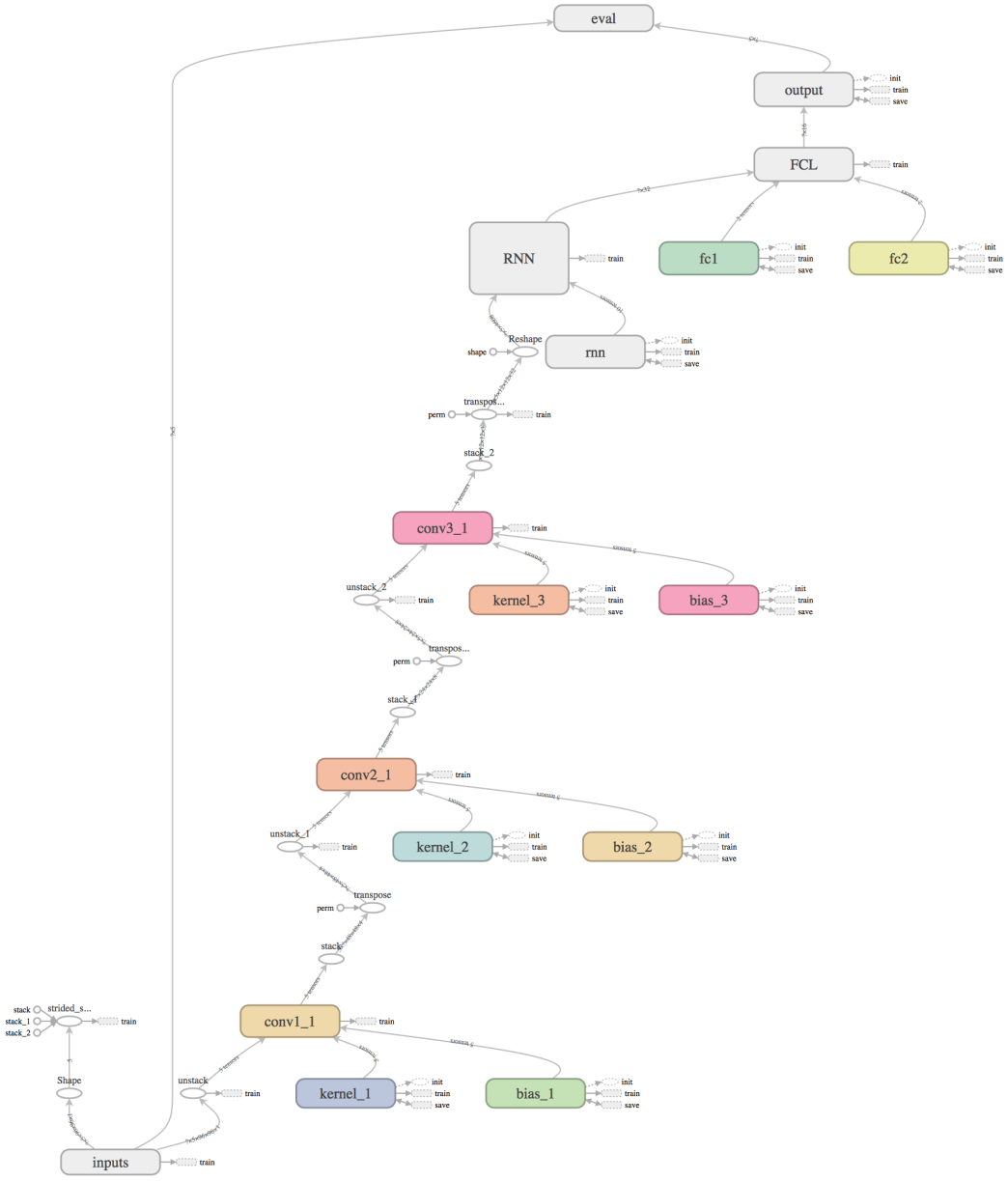
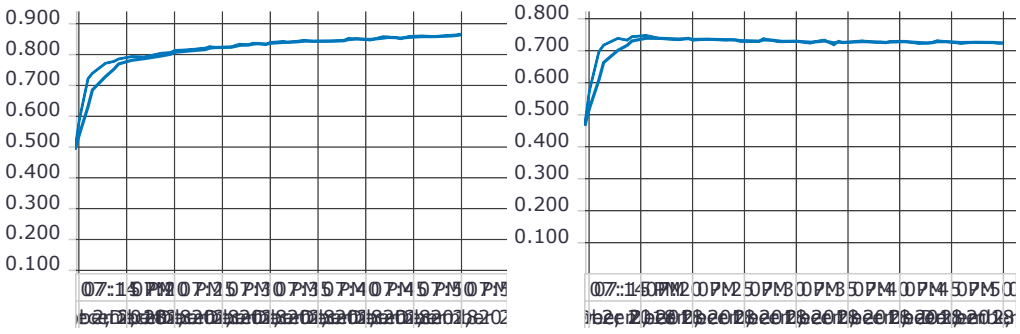


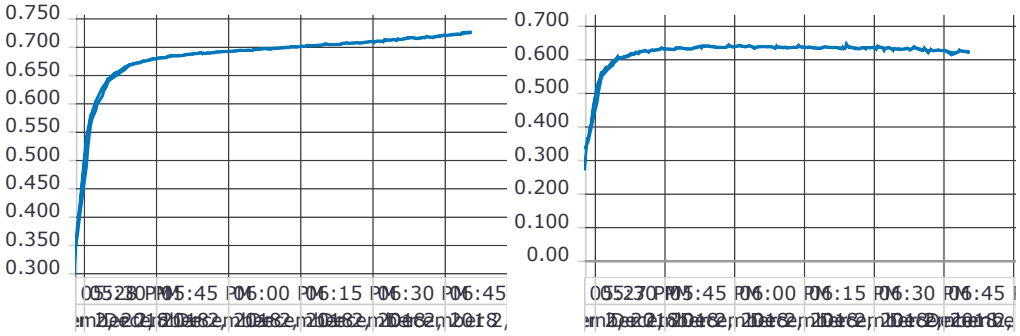
Figure 12: JABnet FlowGraph.

B Appendix B: Learning Curves



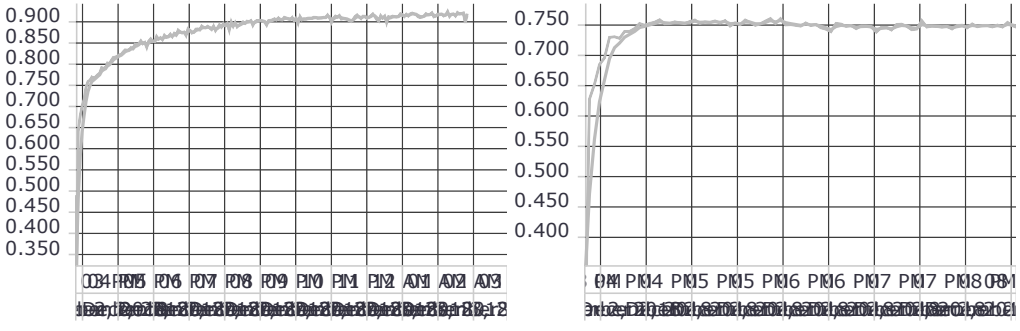
(a) Training Accuracy. (b) Validation Accuracy.

Figure 13: net1 Learning Curves.



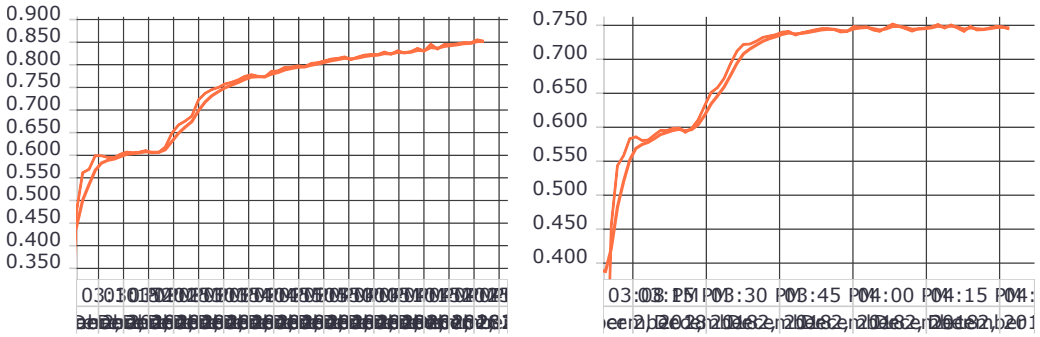
(a) Training Accuracy. (b) Validation Accuracy.

Figure 14: net2 Learning Curves.



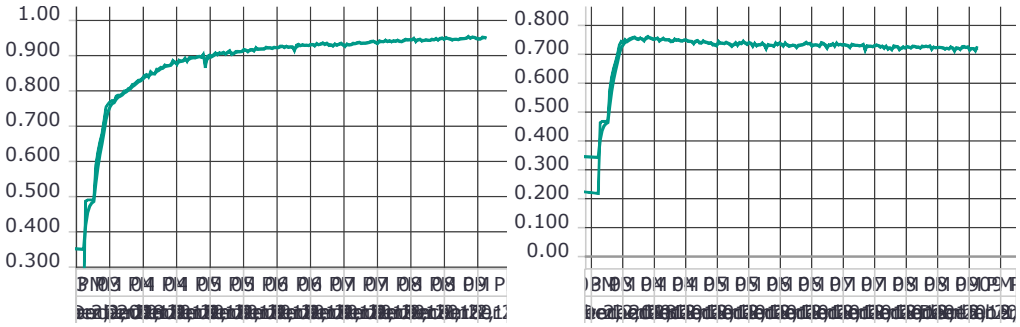
(a) Training Accuracy. (b) Validation Accuracy.

Figure 15: net3 Learning Curves.



(a) Training Accuracy. (b) Validation Accuracy.

Figure 16: net4 Learning Curves.



(a) Training Accuracy. (b) Validation Accuracy.

Figure 17: JABnet Learning Curves.

C Appendix C: Convolutional Kernels and Feature Maps for JABnet

In an effort to visualize at least part of the features learned by the models, this appendix shows the values of the weights used for the convolutional layers of the JABnet model, as well as the resulting feature maps after each convolution and pooling operation.

For the sample feature maps, the `filter_to_image.py` script was used, passing the following state image as a parameter to the model:

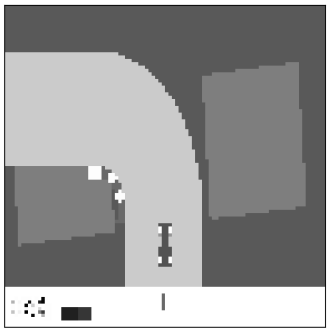
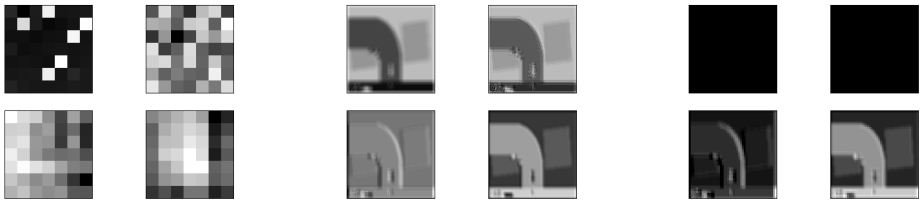


Figure 18: Sample State for the Feature Maps.

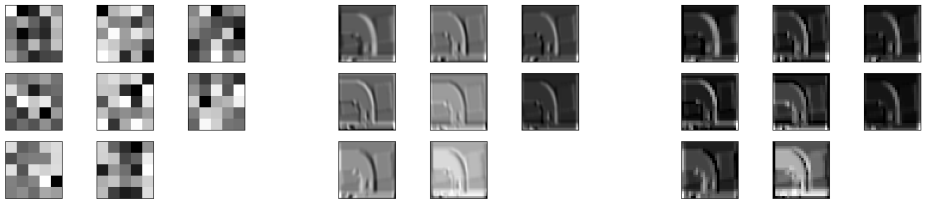
C.1 Convolutional Layer 1



(a) Convolution Kernels. (b) Convolution Feature Maps. (c) MaxPool Feature Maps.

Figure 19: First Convolutional Layer.

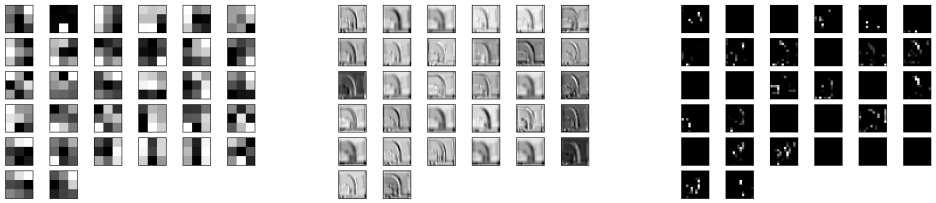
C.2 Convolutional Layer 2



(a) Convolution Kernels. (b) Convolution Feature Maps. (c) MaxPool Feature Maps.

Figure 20: Second Convolutional Layer.

C.3 Convolutional Layer 3



(a) Convolution Kernels. (b) Convolution Feature Maps. (c) MaxPool Feature Maps.

Figure 21: Third Convolutional Layer.