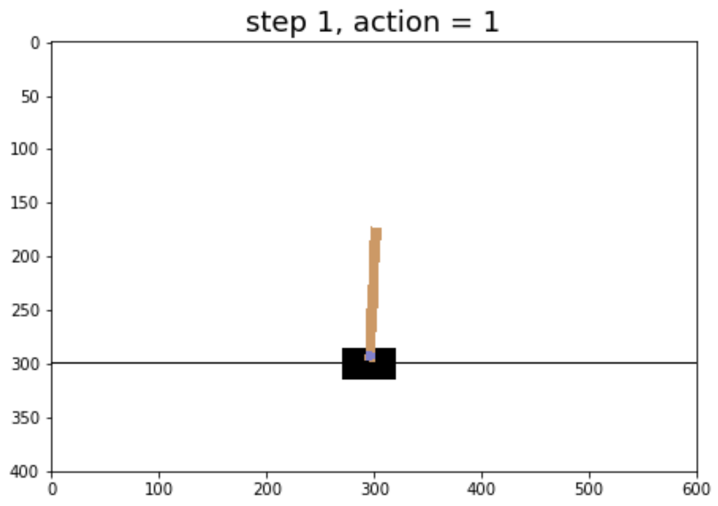
Deep Q-Learning implementation details

Adjusting our basic Q-Learning code

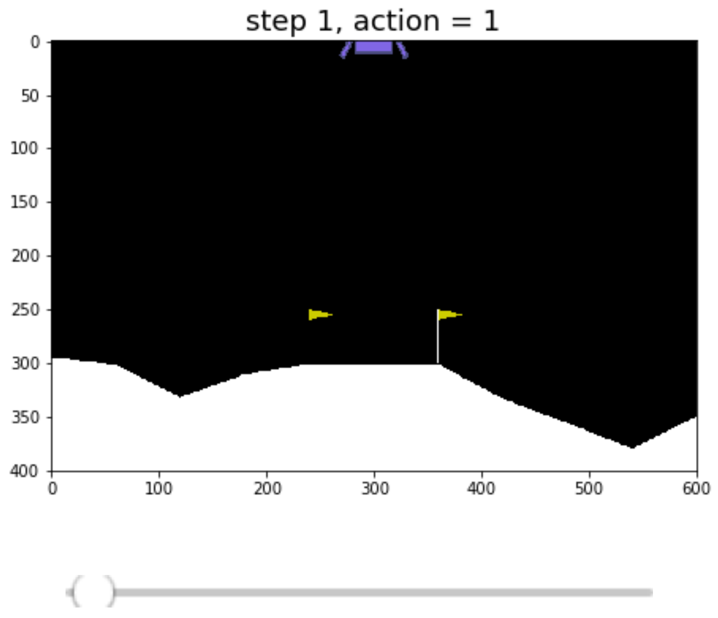
In [83]:
```
animator.animate_run()
```

Out[83]:



In [42]:
```
animator.animate_run()
```

Out[42]:



Review of basic Q-Learning implementation

- with basic Q-Learning (i.e., where Q is a data structure, not a bunch of function approximators) we run each episode by
  - taking a random initial state
  - choosing an action (using the exploration-exploitation trade-off)
  - and repeat taking steps until a goal state is reached or maximum number of steps is taken
- at the t^{th} step we are at a state s_t and take action a_t and update $Q(s_t, a_t)$ using the recursive definition of Q, as

</p>

$$Q\left(s_t,\ a_t\right) = r_t + \gamma \underset{a_{t+1}}{\text{maximum}}\ Q\left(s_{t+1},\ a_{t+1}\right)$$

```
**input** simualtor (containing $f_{\text{system}}$ and reward structure), initialization for $Q$ function, parameter settings
    **for** e = 1...E (the maximum number of episodes)
        Select a initial state $s_1$
        **while** simulation not ended AND maximum iteration count not met
            **Choose action:** $a_t$
            **Transit states:** $s_{t+1},\ \text{done} = f_{\text{system}}\left(s_t, a_t\right)$
            **Get reward:** $r_t$
            **Update Q:** over point $\left(s_t,\ a_t,\ r_t,\ s_{t+1}\right)$
                $Q\left(s_t,a_t\right) = \begin{cases} r_t + \gamma \underset{a_{t+1}}{\text{maximize}}\, Q\left(s_{t+1},a_{t+1}\right) & \text{if done} = \text{False} \\ r_t & \text{if done} = \text{True} \end{cases}$
**output** resolved version(s) of $Q$, history of total episode rewards (for de-bugging / param tuning)
```

- here's a general Q Learning implementation in `Python` for a single episode

In [ ]:
```python
# create random initial state for this episode
state = simulator.reset()

# start episode simulation
total_reward = 0
done = False
while done == False:
    # choose next action
    action = choose_action(state)

    # transition to next state, get associated reward
    next_state,reward,done = simulator.get_movin(action)

    # update Q function
    update_Q(state,next_state,action,reward,done)

    # update total reward from this episode
    total_reward+=reward

    # update current state
    state = copy.deepcopy(next_state)
```

- and each of the individual functions called in this loop
- the 'reset' and 'move' commands are defined by the system / simulator

- next in the inner loop, notice that the `transition` and `reward` steps from pseudo-code are performed in a single line

```
next_state,reward,done = simulator.step(action)
```

- this is commonly how simulators are built (i.e., [OpenAI gym (https://gym.openai.com/)](https://gym.openai.com/) simulator syntax)
- this is often thought of as a 'black-box' operation, whose internals differ from problem to problem

- but remember two things being done in this step
  - the transition $s_{t+1}, \text{done} = f_{\text{system}}\left(s_t, a_t\right)$, and
  - the user-defined reward structure is called to return $r_t = f_{\text{reward}}\left(s_t, a_t, s_{t+1}\right)$
- notice a `done` signal is also returned, if `done = True` the episode has ended, otherwise `done = False`

In [ ]:
```
# update Q function
def update_Q(state,next_state,action,reward,done):
    q_update = reward
    if done == False:
        q_update += gamma*max(Q[next_state][:])
    Q[state][action] = q_update

# exploration-exploitation action selection
def choose_action(state):
    r = np.random.rand(1)
    if r > self.p:  # choose action based on Q (exploit)
        action = np.argmax(self.Q[state,:])
    else:      # choose random action (explore)
        action = np.random.permutation(num_actions)
    return action
```

Basic Deep Q-Learning pseudo/code

- what about the basic Q-Learning procedure must adjust to integrate in machine learning (function approximators)?
- the main Q-Learning loop consists of 4 steps:
  - choose an action
  - transition to the next state based on that action
  - get a reward for this transition
  - update the Q function based on this reward
- what changes when we move to 'deep Q-Learning'?
- *Fundamentally, only the way we update Q* - remember its now a parameterized nonlinear *multi-output regression model*

- now Q is updated via gradient descent step of an appropriate *regression* with 'memory replay'

- that is, this step of the pseudo-code

$$Q\left(s_t,a_t\right) = \begin{cases} r_t + \underset{a_{t+1}}{\text{maximize}}\, Q\left(s_{t+1},a_{t+1}\right) & \text{if done} = \text{False} \\ r_t & \text{if done} = \text{True} \end{cases}$$

- so the pseudo-code, and the `Pythonic` code for the inner loop, *doesn't change*!

- what will change is what happens inside of our **update Q** step, `Pythonically`

```
update_Q(state,next_state,action,reward,done)
```

- our adjustment to `update_Q` consists of *three basic steps*

1) (prior to running episodes) the creation an appropriate set of function approximators / feature transforms / network architecture

2) (during each episode) updating a finite length queue (or window) of trailing tuples (our *memory*), including datapoints from each step of simulation of the form

```
(state,action,reward,next_state,done)
```

3) (to update Q) taking a gradient descent step with respect to (a subset of) tuples from our trailing window to update the parameters of Q

For Step 1)

- Step **1)** will differ slightly depending on what kind of framework you use - e.g., pure Python, Pytorch, Tensorflow, Chainer, etc.,

- regardless of the sort of standard settings you make for your network - e.g., activation function, number of hidden layers, etc, - your
  - *input dimension* to your network needs to equal the dimension of your states
  - *output dimension* to your network needs to equal the dimension of your actions

For Step 2)

- Step **2)** is straightforward, you can use a `list` or any other sort of queue data structure

- at each step of each episode of simulation, you need to load the most recently made tuple

```
(state,action,reward,next_state,done)
```

  into memory, and discard the oldest if you have filled up your maximum allowable memory

For Step 3)

- Step **3)** (to update Q) taking a gradient descent step with respect to (a subset of) tuples from our trailing window to update the parameters of Q
- remember the cost function here is a Least Squares regression
- e.g., over a single datapoint $(s_t,\ a_t,\ r_t,\ s_{t+1})$ remember we replace the fixed update (when Q is a finite data structure)

$$Q\left(s_t, a_t\right) = r_t + \gamma \underset{k=1,...,A}{\text{maximize}}\ Q\left(s_{t+1}, a_{t+1}\right)$$

with the desire

$$Q\left(s_t, a_t\right) \approx r_t + \gamma \underset{k=1,...,A}{\text{maximize}}\ Q\left(s_{t+1}, a_{t+1}\right)$$

- the right hand side above is a *fixed quantity*, lets denote it q_t, then associated Least Squares cost is

$$\left(Q\left(s_t, a_t; \Theta_{a_t}\right) - q_t\right)^2$$

- note here both s_t and a_t are fixed (they're already chosen) values as well
- we take a gradient descent step to update the parameters Θ_{a_t} - the parameters of the model associated with action a_t
- more generally over a set of points in Ω our associated cost is

$$\sum_{(s_t,\ a_t,\ r_t,\ s_{t+1}) \in \Omega} \left(Q\left(s_t, a_t; \Theta_{a_t}\right) - q_t\right)^2$$

- so our pseudo-code does not change much from its original form with our discrete state-space Q-Learning

Basic Deep Q Learning algorithm

```
input simualtor (containing $f_{\text{system}}$ and reward structure), initialization for $Q$ function, parameter settings
    Initialize data queue: $\Omega = \{\}$
    for e = 1...E (the maximum number of episodes)
        Select a initial state $s_1$
        while simulation not ended AND maximum iteration count not met
            Choose action: $a_t$
            Transit states: $s_{t+1} = f_{\text{system}}\left(s_t, a_t\right)$
            Get reward: $r_t$
            Update queue: add new point $\Omega \longleftarrow \Omega \cup \left(s_t,\ a_t,\ r_t,\ s_{t+1},\ \text{done}\right)$, remove oldest point if queue full
            Update Q: over (subset of points in) queue $\Omega$ by taking a gradient descent step in the cost
                $\sum_{\left(s_t,\ a_t,\ r_t,\ s_{t+1}\right) \in \Omega} \left(Q\left(s_t, a_t; \Theta_{a_t}\right) - q_t\right)^2$
output resolved version(s) of $Q$, history of total episode rewards (for de-bugging / param tuning)
```

Deep Q Learning basic enhancements

- like our move from the basic tabular Q-Learning algorithm to more sophisticated versions (including e.g., an exploit/explore parameter), we can make some simple adjustments to our deep Q Learning algorithm to improve it (i.e., to speeden up the learning of a proper Q function
- these simple adjustments range from macro-tweaks to the larger pseudo-code, to micro-tweaks to things like our choice of network architecture to optimizer

'Fitted Q-Learning'

- lets briefly discuss a common variation of this basic scheme
- remember that our challenge with Q-Learning: when can we trust Q?
- if we are using exploration-exploitation to choose actions, note that during a *single episode* exploitation actions are taken with respect to different versions of Q *after each action is chosen in an episode*
- when we use a global function approximator - like a fully connected network - then the *entire* shape of the function changes if we re-fit it at a point (even when taking a single gradient descent step)
- so *entire shape* of Q changes from step-to-step, if we update it *during an episode*

- this means that way in which we take exploitation actions - via our optimal control law

$$a_t^\star = \underset{a_t}{\text{argmax}}\ Q\left(s_t, a_t\right)$$

changes every single step during a single episode

- in other words, our optimal control law itself oscillates *inside each episode*

- to stabilize the learning of Q we can take the Q update step out of the episode loop, collecting all data during an episode, and update Q over all this data at once after the episode finishes
- this simple adjustment has its own jargon name - [Fitted Q Learning (http://ml.informatik.uni-freiburg.de/former/_media/publications/rieecml05.pdf)](http://ml.informatik.uni-freiburg.de/former/_media/publications/rieecml05.pdf) as coined by its author - and drastically improves performance (and was the basis for the deep reinforcement resurgance started by Deep Mind)
- note: 'simple' doesn't mean 'obvious', far from it
- notice in addition to moving our `update_Q` step outside the episode loop, we add a single step to store each step's tuple `(state,next_state,action,reward,done)`

Basic Fitted-Q Learning algorithm

```
input simualtor (containing $f_{\text{system}}$ and reward structure), initialization for $Q$ function, parameter settings
    Initialize data queue: $\Omega = \{\}$
    for e = 1...E (the maximum number of episodes)
        Select a initial state $s_1$
        while simulation not ended AND maximum iteration count not met
            Choose action: $a_t$
            Transit states: $s_{t+1} = f_{\text{system}}\left(s_t, a_t\right)$
            Get reward: $r_t$
            Update queue: add new point $\Omega \longleftarrow \Omega \cup \left(s_t,\ a_t,\ r_t,\ s_{t+1},\ \text{done}\right)$, remove oldest point if queue full
        Update Q: over (subset of points in) queue $\Omega$ by taking a gradient descent step in the cost
            $\sum_{\left(s_t,\ a_t,\ r_t,\ s_{t+1}\right)\in\Omega} \left(Q\left(s_t, a_t; \Theta_{a_t}\right) - q_t\right)^2$
output resolved version(s) of $Q$, history of total episode rewards (for de-bugging / param tuning)
```

- this kind of simple change can make a big impact on performance, the rate at which Q is properly resolved
- it is easily generalized to e.g., updating Q every e episodes
- we can still use memory replay, and since we use episode as increment we also change memory length to keep a defined number of episodes of past simulations instead of a defined number of total steps
- why not do this with regular Q-Learning (when Q can be represented as a discrete data structure)?

A second practical Q stabilizing idea: fix the Q per episode wherever it appears as a *target*

- an alternative to fixing Q per episode - fix it only where it is used as to provide trusted long term reward (where it is used as 'target')
- in an episode Q arises in two locations - when we choose an action via exploitation, and when we make our Q update i.e.,
  - i.e., when we use it as a trusted optimal control law to transition in-episode

$$a_t = \underset{k=1,...,A}{\text{argmax}} \ Q\left(s_t, \alpha_k\right)$$

  - and when we update its parameters via multi-output regression step

$$Q\left(s_t, a_t\right) \approx r_t + \gamma \underset{k=1,...,A}{\text{maximize}} \ Q\left(s_{t+1}, a_{t+1}\right)$$

- in both instances the Q on the right hand side is trusted to provide optimal long term reward / control
- so we fix the parameters here at the previous episode's update

- In other words, if Θ^- is last episode's learned parameters (from out `update_Q` step), then our two equations are (exposing weights of Q throughout)

$$a_t = \underset{k=1,...,A}{\text{argmax}} \ Q\left(s_t, \alpha_k; \Theta^-\right)$$

$$Q\left(s_t, a_t; \Theta\right) \approx r_t + \gamma \underset{k=1,...,A}{\text{maximize}} \ Q\left(s_{t+1}, a_{t+1}; \Theta^-\right)$$

- Then at the end of episode update parameters of Q on the right hand side of each update equation above
- Call end of episode updated parameters Θ, then replace $\Theta \longleftarrow \Theta^-$
- $Q\left(s_{t+1}, a_{t+1}; \Theta^-\right)$ often referred to as a *target network* or *target Q function* and was introduced in 2015 (https://www.nature.com/articles/nature14236)

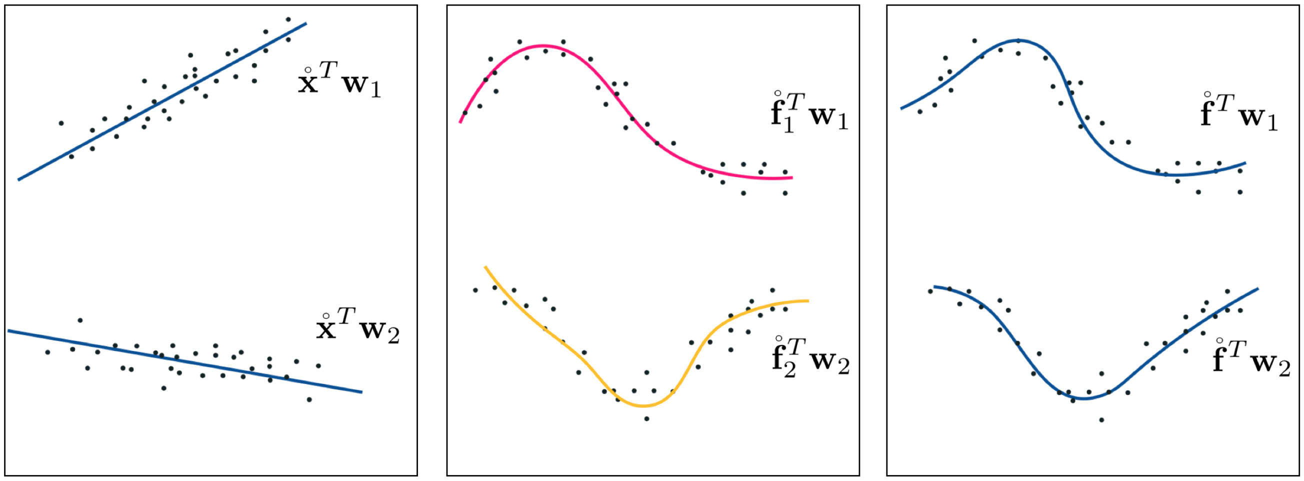
Using a shared architecture for standard supervised learning

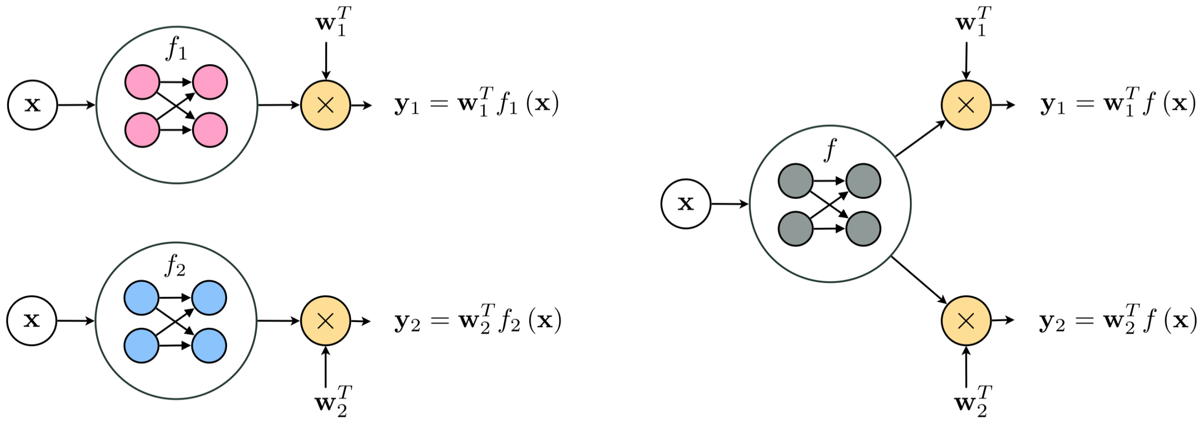
- in our discussions thus far we have assumed we use a single function approximator / feature transform / network for *each action / column of* Q
- we can, however, use a *single network* to reprent *all columns* at once
- this is the standard 'multiple-output' regression style network architecture commonly with neural networks (i.e., when "doing deep learning")

- its a very convenient choice when using fully connected networks (https://jermwatt.github.io/machine_learning_refined/notes/13_Multilayer_perceptrons/13_2_Multi_layer_perceptrons.html) (as we'll briefly review)
- we can adapt it - with one slight but important variation - to the general Q-Learning scenario
- this combination (shared network architecture + Q-Learning) is typically what folks refer to as "deep Q-Learning"

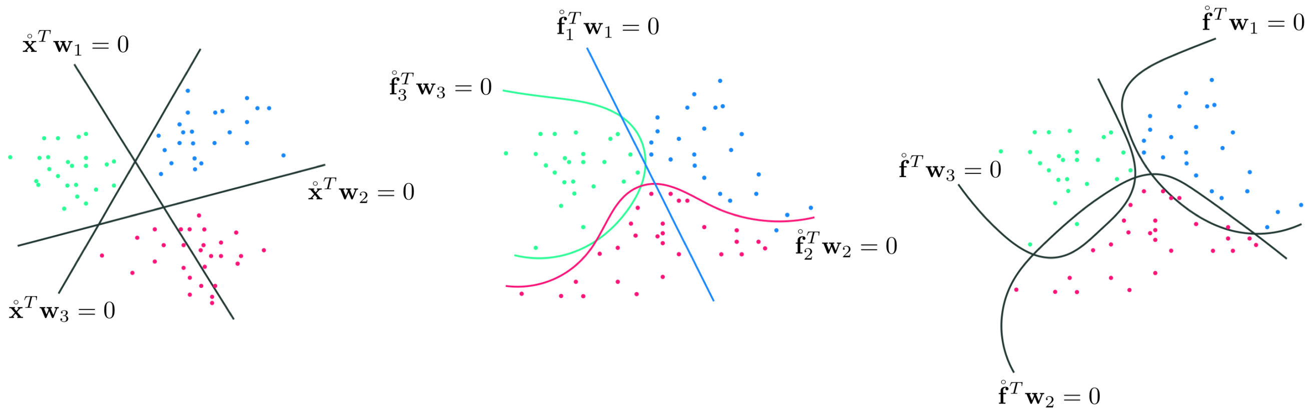
- when performing *linear* supervised learning (multi-class classification (https://jermwatt.github.io/machine_learning_refined/notes/7_Linear_multiclass_classification/7_3_Perceptron.html) and multi-output regression (https://jermwatt.github.io/machine_learning_refined/notes/5_Linear_regression/5_6_Multi.html)) you assign a set of unique weights to each classifier / regressor
- you can then either tune each set of unique weights separately, or together (e.g., with classification you can perform One-versus-All classification (https://jermwatt.github.io/machine_learning_refined/notes/7_Linear_multiclass_classification/7_2_OvA.html), or perform multi-class logistic regression (https://jermwatt.github.io/machine_learning_refined/notes/7_Linear_multiclass_classification/7_3_Perceptron.html))
- when you extend this concept to the nonlinear setting you can do the same thing, but further you can share a single nonlinearity across *all the learners*, which is most naturally done when using fully connected nonlinearities

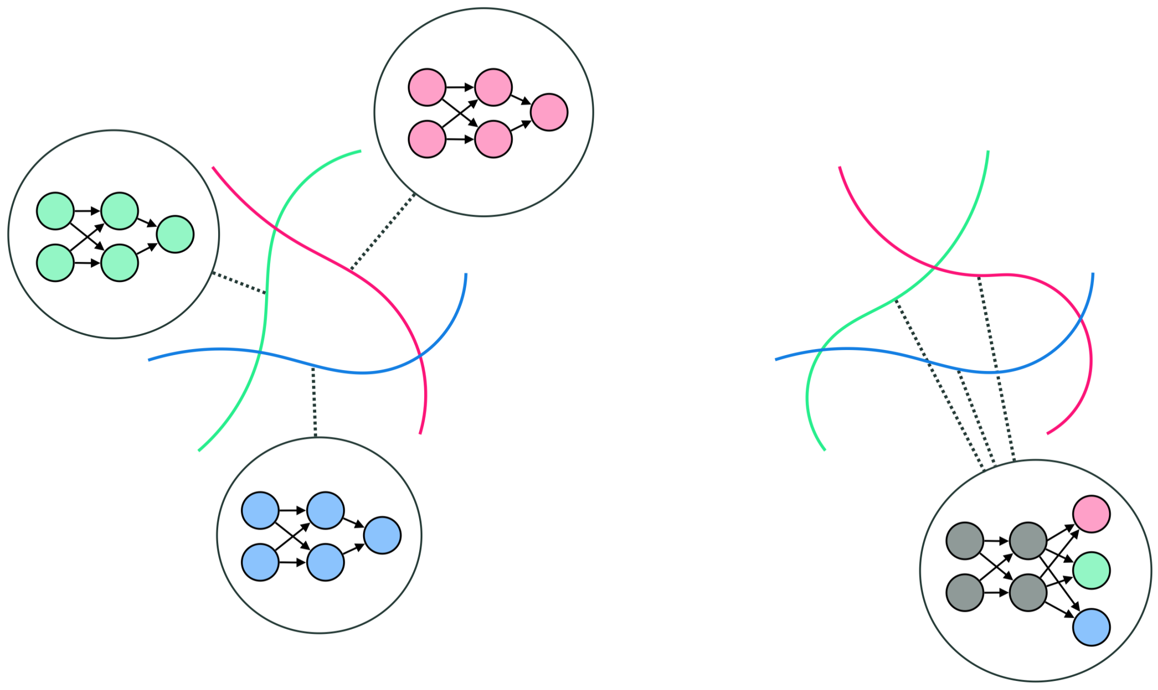
- with nonlinear multi-output regression (https://jermwatt.github.io/machine_learning_refined/notes/10_Nonlinear_intro/10_3_MultReg.html), the situation looks like this
- say you have two outputs to predict, you can choose a distinct nonlinearity for each output f_1 and f_2, or one *shared* nonlinearity f for both outputs
- in the latter case this means that *both regressors share the same nonlinearity, but each has its own unique final linear combination weights*
- here's what this looks like with regression




$\mathbf{x}$
f_1
$\mathbf{w}_1^T$
$\times$
$\mathbf{y}_1 = \mathbf{w}_1^T f_1(\mathbf{x})$
f_2
$\mathbf{w}_2^T$
$\mathbf{y}_2 = \mathbf{w}_2^T f_2(\mathbf{x})$
f
$\mathbf{y}_1 = \mathbf{w}_1^T f(\mathbf{x})$
$\mathbf{y}_2 = \mathbf{w}_2^T f(\mathbf{x})$

- here's what it looks like with $C = 3$ class classification





- why share a single nonlinear architecture?
- its by far the more common way of doing things when using neural network nonlinearities because it drastically "simplifies the UI":
    - **parameter-effectient:** fully connected networks are highly parameterized, and this allows you to scale more gracefully with the number of outputs / classes
    - **convenient:** instead of worrying about how to correctly choose architectures for each output (i.e., how to cross-validate each output network) you just worry about how to properly choose the network in its entirety

Nonlinear mulit-output regression using a shared network

- see [reinforcement learning docker image (https://paper.dropbox.com/doc/Running-reinforcement-docker-image--AdJDQoFixnghaWm2tkuEFYM_Ag-jArlqs3n7ctBmN7phAcv7)](https://paper.dropbox.com/doc/Running-reinforcement-docker-image--AdJDQoFixnghaWm2tkuEFYM_Ag-jArlqs3n7ctBmN7phAcv7) associated with this class for examples of how to build shared architecture nonlinear regressors using autograd, PyTorch, and Keras / Tensorflow

Sharing a network architecture for Q Learning

- in the same way here using a single network to represent all actions / columns of Q makes fine-tuning chores simpler
- i.e., we decide / test one architecture with one setting for number of hidden layers / units / etc., as opposed to one-per-column
- however unlike the use of networks with multi-output regression, here doing this introduces an obvious problem (that has an equally obvious solution)

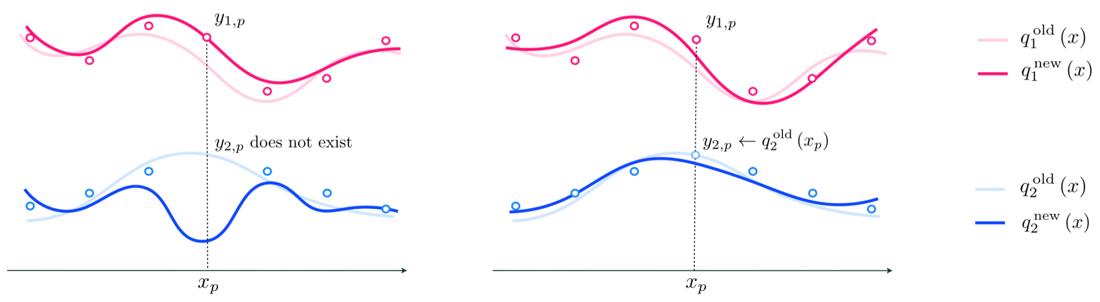
- there one catch here: what if you are missing one output from an input/output pair, and you try to tune the weights of the *shared* architecture?
- when does this kind of thing happen? when we use a shared architecture for Q Learning. think about it, we will generate tuples of the form

```
(state,next_state,action,reward,done)
```

  during one step of one episode, and then go on to update Q using it (along with a trailing sample of past tuples ala *memory replay*)

- each such tuple contains **only one** output - given by the value of `action` - meaning when we update our parameters we will be missing *several output values*
- think about it: if we then update our *shared parameters* belonging to our *shared architecture* based on only one of our outputs, we aren't restricting our other regressors near this input/output point
- so since we do not tell our shared architecture anything about what shape needs to be maintained at our given input with any regressors that do not have an associated output, they are free to change wildly (as shown in the left panel below)

- the practical solution: clamp all regressors to their current outputs by creating *phantom output values* for any regressor that does not have an associated output (as shown in the right panel below)
- this gives *every regressor* something to regress against - and prevents those without output values from going crazy
- it looks like this



- thus if using a shared architecture, this means we must construct *phantom outputs* for any regressor missing outputs in our queue during a parameter update

Examining the entire implementation

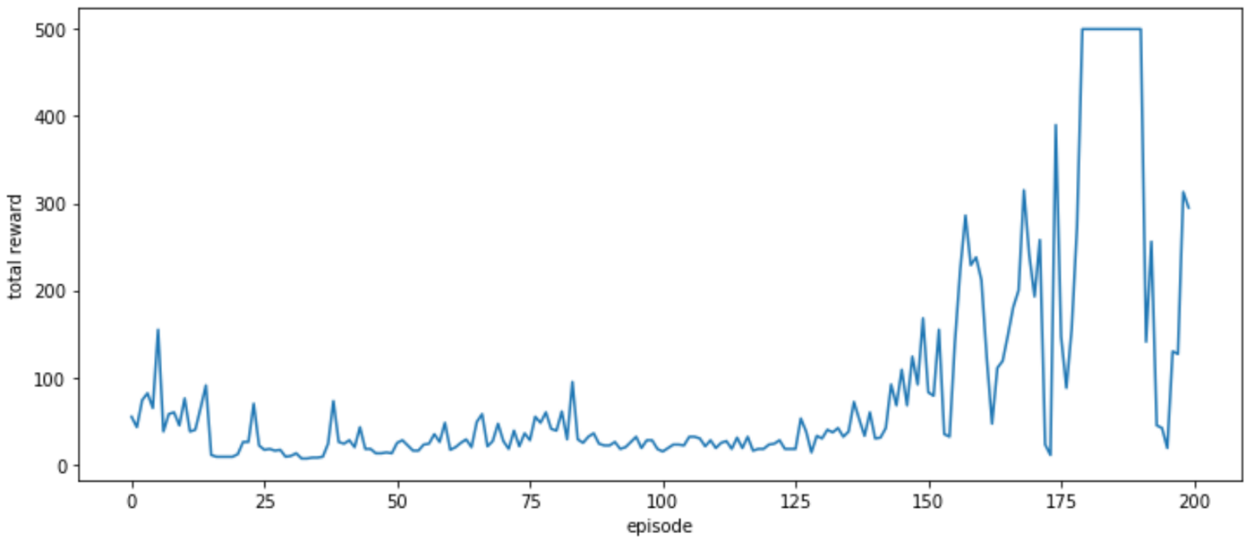
- see the reinforcement learning docker image (https://paper.dropbox.com/doc/Running-reinforcement-docker-image--AdJDQoFixnqhaWm2tkuEFYM_Ag-jArlqs3n7ctBmN7phAcv7) associated with this class for complete examples of this basic implementation of deep Q Learning in autograd, PyTorch, and Keras / Tensorflow

Practical matters

- we will talk in much more detail about practical issues associated with creating an effective Deep Q Learner, but for now a few important points (e.g., for homework)

Deep Q-Learning can be highly unstable

- below a run for the **cartpole** problem



Defining a reward structure can be very difficult

- even with small problems (e.g., gridworld) defining rewards carefully is important
- e.g., with gridworld if goal state's reward is not significantly different than standard or hazard states learning will take much longer
- but remember now - we're using function approximators too! And performing many many rounds of **online regression**
- thus if reward values are similar for both *good* and *bad* states to be in, it will be *very difficult* for Q function to resolve