CS 234 Session III
RL with Function Approximation

Content

- Lecture Review
  - Motivation & SGD review
- Neural Network Basics
  - Definition
  - Backpropagation
  - Fully-connected networks
  - Convolutional neural networks

Lecture Review

Review: Why use Function Approximation?

PROS

- Efficient state value representation
- Generalization to unseen states

CONS

- Not exact (i.e. can have error even for states visited many times)
- Convergence guarantees are lost

Review: Stochastic Gradient Descent

- Loss function: $J(\mathbf{w})$
  - Parametrized by and differentiable w.r.t its parameters
- Training data: $\{(\mathbf{x}_i, y_i)\}_{1:n}$
  - Set of training examples

Objective? Iteratively find parameters that minimize the loss!

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w})$$

How? By following the (negative) gradient $-\nabla J(\mathbf{w})$

Review: Stochastic Gradient Descent

- Gradient of loss

$$\nabla J(\mathbf{w}) = \langle \frac{\partial J(\mathbf{w})}{\partial \mathrm{w}_1}, \ldots, \frac{\partial J(\mathbf{w})}{\partial \mathrm{w}_n} \rangle$$

- During each iteration, update the parameter vector

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla J(\mathbf{w})$$

- Learning rate α controls the rate of update

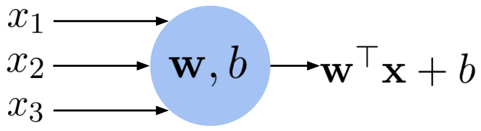
Neural Network Basics

What is a Neural Network?

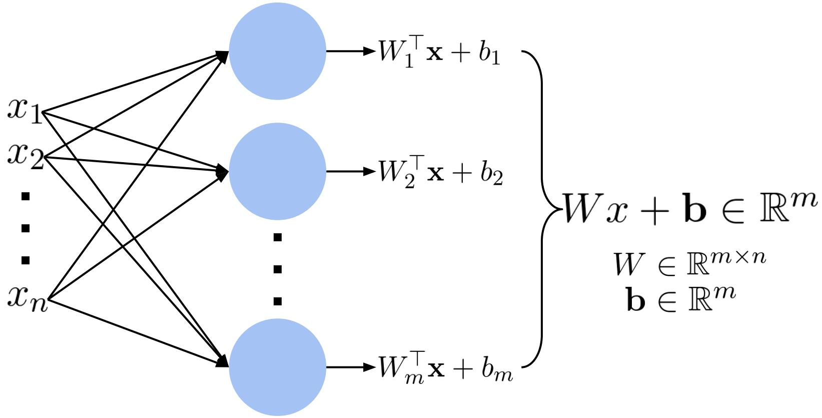
- A neural network is a **function** that consists of:
  - **Neurons** that take values and produce an output
  - **Weights** that control how values are carried between neurons
- Neurons are grouped into **layers**:
  - Input layer
  - Hidden layer(s)
  - Output layer

A Deeper Look at a Neuron

- A neuron can have many inputs, but produces one value



- When you have many neurons, it becomes a (**fully-connected**) layer

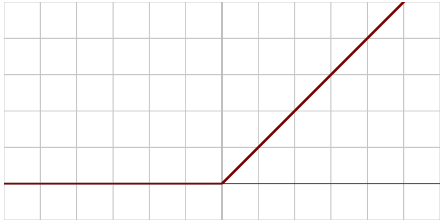


More Details

- No matter how many layers are stacked, it's still a linear function!
  - We apply non-linear function to the output of each layer
- Starting from the input
  - Output of hidden layer 1: $\mathbf{h}_1 = f(W_1\mathbf{x} + b_1)$
  - Output of hidden layer 2: $\mathbf{h}_2 = f(W_2\mathbf{h}_1 + b_2)$
    $$= f(W_2 f(W_1\mathbf{x} + b_1) + b_2)$$
  - Output of hidden layer 3: $\mathbf{h}_3 = f(W_3\mathbf{h}_2 + b_3)$
  - where f is an element-wise **activation function**
- In Tensorflow: `tf.layers.dense`

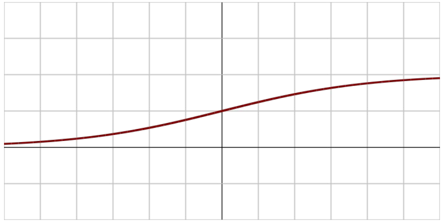
Activation Functions

- Generally a monotonic, differentiable, non-linear function from $\mathbb{R} \to \mathbb{R}$
- Applied to each element of a vector, matrix, tensor.
- Lots of options to choose from:
  - ReLU, sigmoid, tanh, leaky ReLU, SoftPlus, ELU, SELU ...
- Examples:



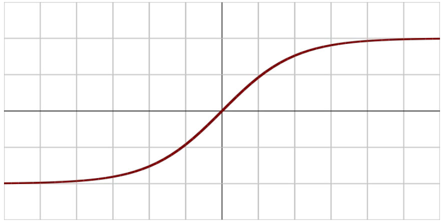
$\mathrm{ReLU}(x) = \max\{x, 0\}$

`tf.nn.relu`



$\sigma(x) = \frac{1}{1+\exp(-x)}$

`tf.nn.sigmoid`



$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

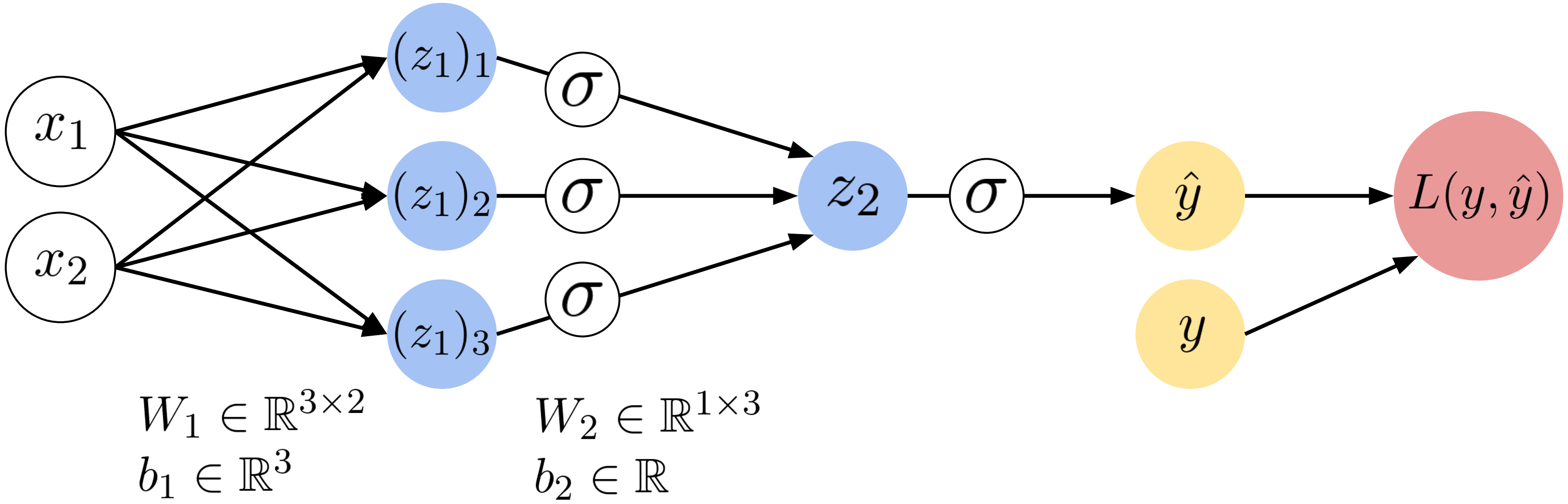
`tf.nn.tanh`

Loss Functions

- Measures the quality of network predictions → problem dependent
- For prediction $\hat{y}$ and the expected (true) value y
- For regression:
  - L1 loss: $||y - \hat{y}||_1$
  - L2 loss: $||y - \hat{y}||_2$
- For classification:
  - Cross-entropy loss: $-\sum_{i=1}^{k} y_i \log \hat{y}_i$
    - Binary case: $-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$

Backpropagation Example

- Taking the gradient of a neural network is done **layer-by-layer**
- Example: 1-hidden layer neural network for binary cross-entropy loss



Backpropagation Example

- Notation
  - Layer 1 (pre-activation): $z_1 = W_1 x + b_1$
  - Layer 1 (post-activation): $a_1 = \sigma(z_1)$
  - Layer 2 (pre-activation): $z_2 = W_2 \cdot \sigma(W_1 x + b_1) + b_2$
  - Network output: $f(x) = \sigma(z_2) = \sigma(W_2 \cdot \sigma(z_1) + b_2)$
  - Binary cross-entropy loss: $L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$

Backpropagation Example (cont'd)

- Step 1: loss → layer 2 output: $\frac{\partial L}{\partial \hat{y}} = \frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}}$
- Step 2: layer 2 output → z_2 : $\frac{\partial \hat{y}}{\partial z_2} = \hat{y}(1-\hat{y})$
- Step 3: z_2 → layer 2 weights: $\frac{\partial z_2}{\partial W_2} = a_1$ $\frac{\partial z_2}{\partial b_2} = 1$
- Step 4: z_2 → layer 1 output: $\frac{\partial z_2}{\partial a_1} = W_2$

Backpropagation Example (cont'd)

- Step 5: layer 1 output → z_1: $$\frac{\partial a_1}{\partial z_1} = \mathrm{diag}(a_1 \odot (1 - a_1))$$
- **Step 6: loss → layer 1 weights:** $$\frac{\partial L}{\partial W_1} = \left(\frac{\partial L}{\partial z_1}\right)^\top x^\top$$ $$\frac{\partial L}{\partial b_1} = \left(\frac{\partial L}{\partial z_1}\right)^\top$$
- Now we apply chain rule!

Backpropagation Example (cont'd)

- Layer 2 gradients:

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} = \left(\frac{1-y}{1-\hat{y}} - \frac{y}{\hat{y}} \right) \cdot \hat{y}(1-\hat{y}) = \hat{y} - y$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial W_2} = (\hat{y} - y) \cdot a_1$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial b_2} = \hat{y} - y$$

Backpropagation Example (cont'd)

- Layer 1 gradients:

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1} = (\hat{y} - y) \cdot W_2 \cdot \operatorname{diag}(a_1 \odot (1 - a_1))$$

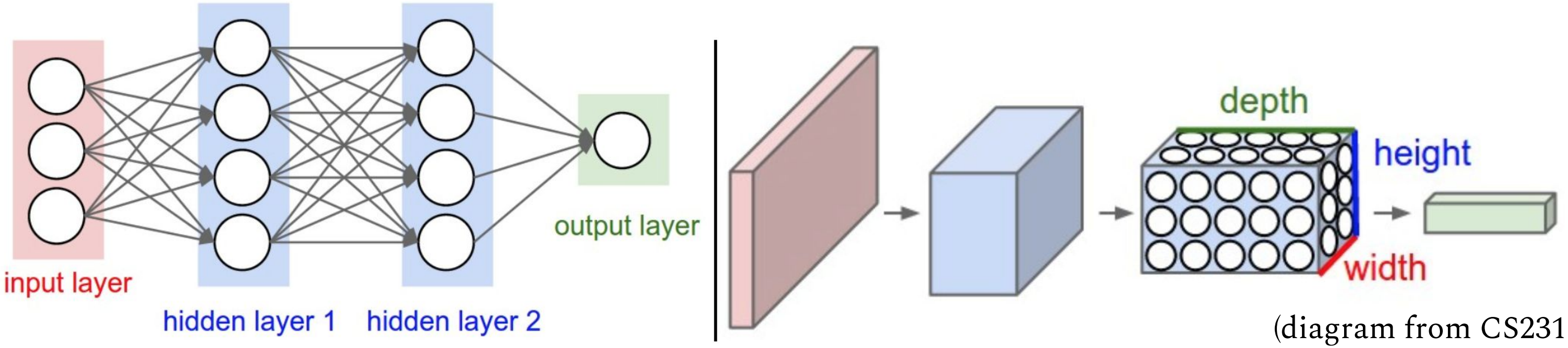
$$\frac{\partial L}{\partial W_1} = \left(\frac{\partial L}{\partial z_1}\right)^\top x^\top = (\hat{y} - y) \cdot \operatorname{diag}(a_1 \odot (1 - a_1)) \cdot W_2^\top \cdot x^\top$$

$$\frac{\partial L}{\partial b_1} = \left(\frac{\partial L}{\partial z_1}\right)^\top = (\hat{y} - y) \cdot \operatorname{diag}(a_1 \odot (1 - a_1)) \cdot W_2^\top$$

- Do the shapes match?

Convolutional Neural Network (CNN)

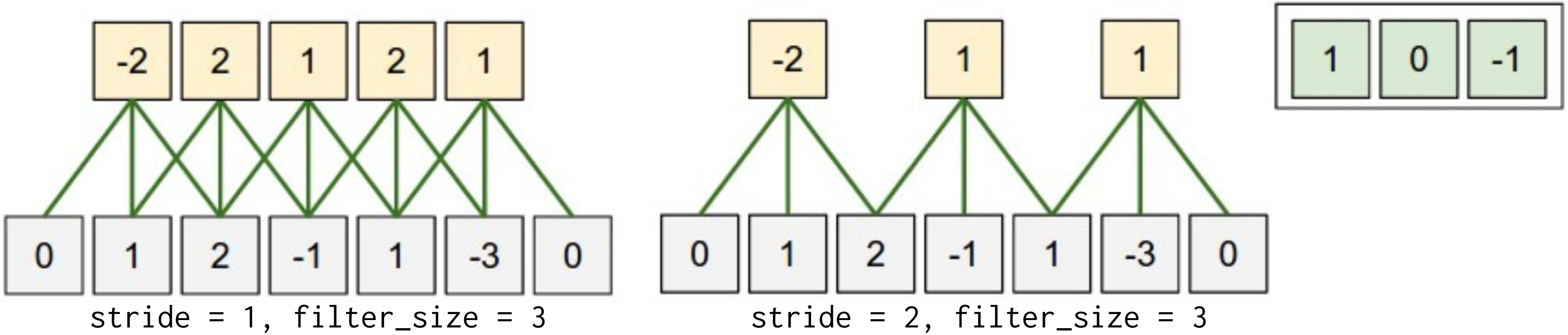
- A type of layer called **convolutional layer** plays well with 2D input.
- It computes the output (which is also 2D) by sliding over its *filter (kernel)* over the input, taking the sum of element-wise products.
- Much fewer parameters than fully-connected layers.
- A simple CNN might look like this: x → conv → conv → conv → FC



(diagram from CS231N notes)

Conv layer in 1D

- Let's take a look at a simpler case of 1D input

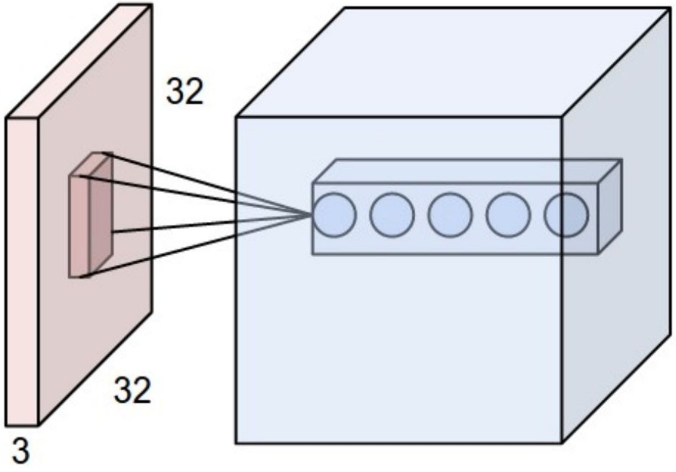


- **Filter (kernel) size** controls how wide the filter is
- **Stride** controls how much move the filter by after each step

(diagram from CS231N notes)

Volumes and Depths

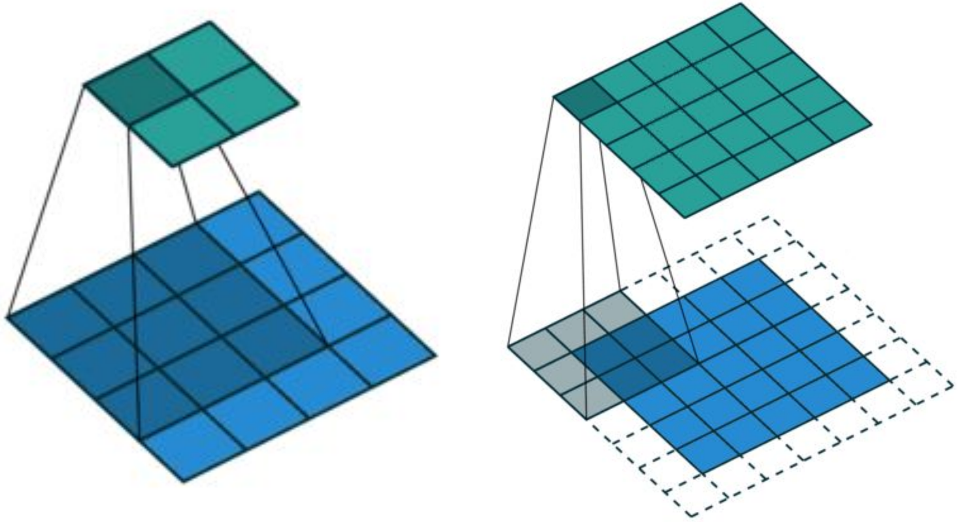
- Same idea, except everything is 2-dimensional now.
- For color images, each pixel has three color channels, so the input is actually a 3D tensor of shape (H, W, 3)!
- Thus, conv layers take 3D input **volumes** and produces output 3D volumes.
- Size of the last axis (color channel) is the **depth** of a volume.
- Filter is also 3D, with the same depth as the input volume.



(diagram from CS231N notes)

Padding

- For any filter size > 1, the output size is smaller
- We can pad the width and height (but **not** depth) of the input volume!
- Usually zero-padded
- Allows us to control the output size of a conv layer.

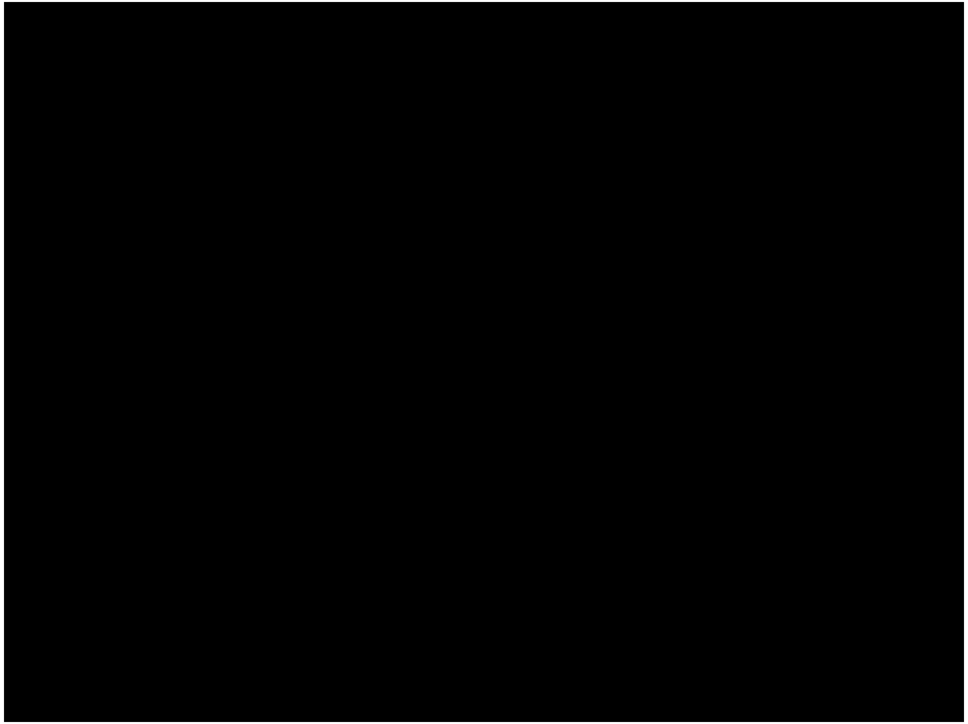


(animation by Vincent Dumoulin)

Finally... Conv layer in 2D

- Combining all of these, a conv layer has the following parameters:
  - **Stride**: controls how much the filter moves by
  - **Padding**: controls how much the input volume is extended by around its edges
  - **Filter size**: controls the width and height of the filter (remember: depth is always equal to the input depth)
- The 2D slice through a specific depth is called a **feature map**.
  - 3D volume = stack of feature maps
- In Tensorflow: `tf.layers.conv2d`

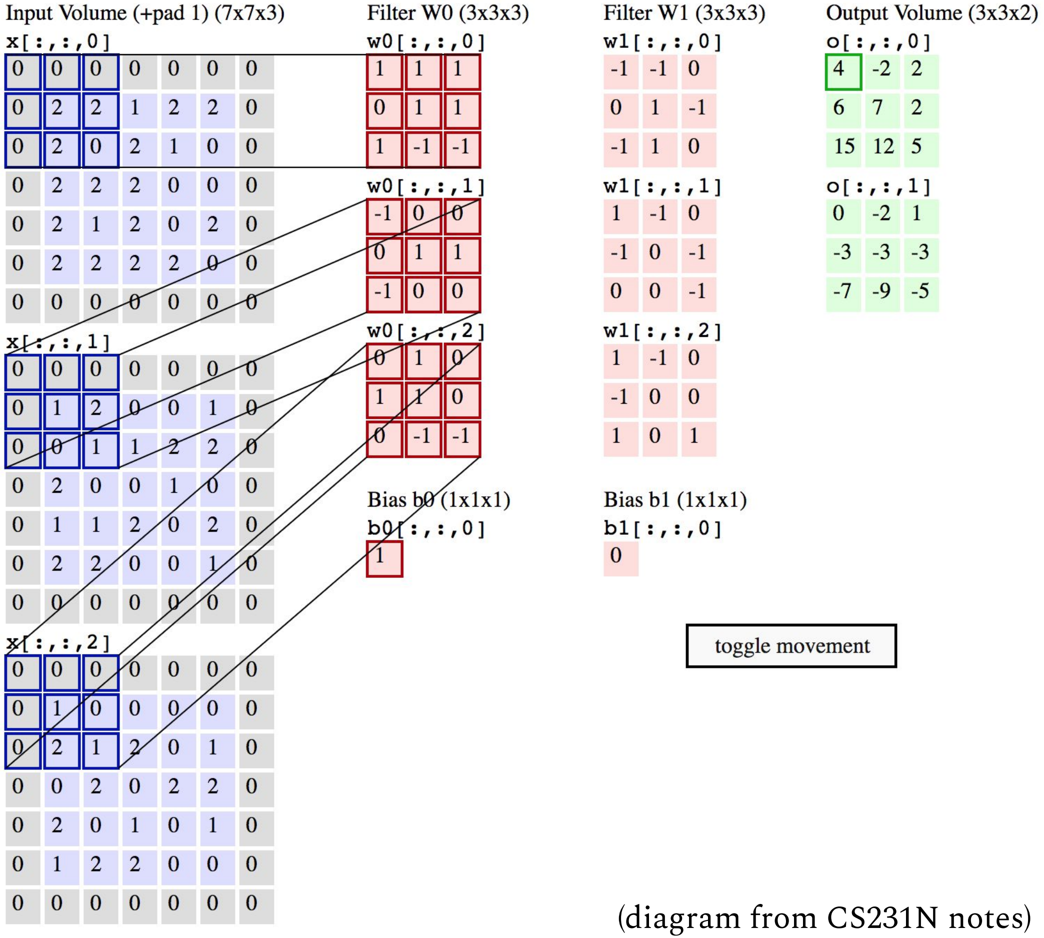
Conv Layer Animation



(video from CS230 lecture notes)

Conv Layer Example

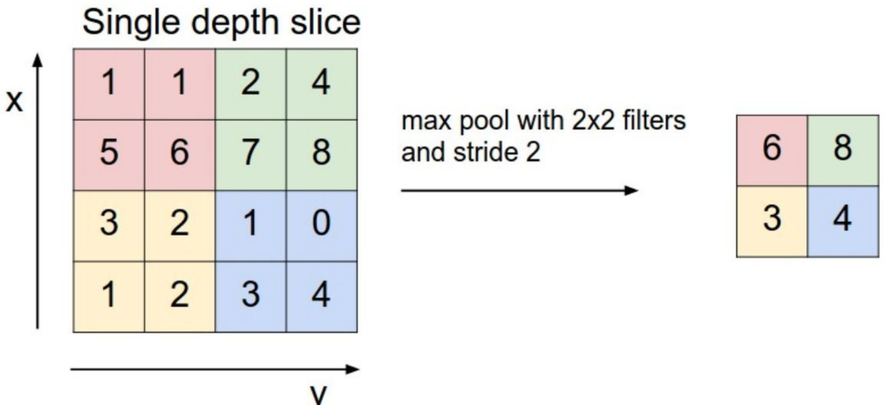
- Input shape: (5, 5, 3)
  - Height = 5
  - Width = 5
  - Depth = 3
- Padding: 1
- Conv layer
  - Filter size = (3, 3)
  - Filter depth = 3
  - Output channels = 2
  - Stride = 2
- Output shape: (3, 3, 2)



(diagram from CS231N notes)

Pooling Layer

- Another way to shrink the dimensions of output volume.
- Usually applied immediately following a conv layer.
- Pooling layers simplify / compress the information in the output from a conv layer by **downsampling** the input volume.
- Max pooling: takes the maximum of the numbers in the receptive field.
- No parameter to be trained!



(diagram from CS231N notes)

Parameter Efficiency

- The same filter is **shared** across the entire input volume.
    - All the neurons in an output feature map detect the same feature, just at **different locations** in the input image.
    - Takes advantage of the translational invariance of image features.
- Example layer sizes: (32 x 32 x 3) → (16 x 16 x 10) → 10
    - Fully-connected network: 3072 * 2560 + 2560 * 10 = ~7.9M
    - CNN with 5x5 kernel & 2x2 pooling: 5 * 5 * 3 * 10 + 2560 * 10 = ~26K
    - About 300x fewer parameters than fully-connected network!

TF References & Further Reading

- **READ THE DOCUMENTATION!**
- **KEYWORD ARGUMENTS ARE IMPORTANT**
- Backpropagation: CS224N notes
- Fully-connected ("dense") layer: `tf.layers.dense`
- Convolutional layer (2D): `tf.layers.conv2d`
- Activation functions: ReLU(`tf.nn.relu`), Sigmoid (`tf.nn.sigmoid`)
- Other useful techniques:
  - Batch Normalization: (`tf.layers.batch_normalization`)
  - Regularization: CS231N notes
    - L1 / L2 regularization, Dropout

Questions?

Review: Assumptions for VFA

Assumptions

- Feature function: $x(s) = [x_1(s) \; x_2(s) \; \dots \; x_n(s)]$
- Linearity: $\tilde{V}(s) = x(s)^\top \mathbf{w} = \sum_i x_i(s) \mathrm{w}_i$
- L2 loss: $J(\mathbf{w}) = \mathbb{E}_\pi \left[(V(s) - \tilde{V}(s))^2 \right]$
- In practice, these assumptions are often not true

Review: Monte Carlo Policy Evaluation with VFA

Algorithm 1 Monte Carlo Linear Value Function Approximation for Policy Evaluation

Initialize $\mathbf{w} = 0$, $Returns(s) = 0\ \forall s$, $k = 1$

loop

  Sample k-th episode $(s_{k1}, a_{k1}, r_{k1}, s_{k2}, \ldots, s_k, L_k)$ given π

  for $t = 1, \ldots, L_k$ **do**

    if first visit to (s) in episode k **then**

      Append $\sum_{j=t}^{L_k} r_{kj}$ to $Return(s_t)$

      $\mathbf{w} \leftarrow \mathbf{w} + \alpha(Return(s_t) - \hat{v}(s_t, \mathbf{w}))x(s_t)$

  $k = k + 1$

What would the update rule for TD(0) policy evaluation look like?

Review: Control with VFA

- As explained last week, we need Q(s,a) for model-free control

Algorithm 5 Q-Learning withϵ-greedy exploration

procedure Q-LEARNING(ϵ, α, γ)
    Initialize $Q(s, a)$ for all $s \in S, a \in A$ arbitrarily except $Q(terminal, \cdot) = 0$
    $\pi \leftarrow \epsilon$-greedy policy with respect to Q
    for each episode **do**
        Set s_1 as the starting state
        $t \leftarrow 1$
        loop until episode terminates
            Sample action a_t from policy $\pi(s_t)$
            Take action a_t and observe reward r_t and next state s_{t+1}
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))$
            $\pi \leftarrow \epsilon$-greedy policy with respect to Q (policy improvement)
            $t \leftarrow t + 1$
    return Q, π

- Analogously, update rule changes to:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha(G_t - \tilde{Q}(s_t, a_t))\nabla\tilde{Q}(s_t, a_t)$$

$$= \mathbf{w} - \alpha(G_t - \tilde{Q}(s_t, a_t))x(s_t, a_t)$$

Back to Value Function Approximation

- How do we represent the value function Q(s,a) with a neural net?
  - Same as linear function approximation (with SGD)
  - Tensorflow computes the gradient for us!
  - What should be the dimension of the output?
- CNN works well for Atari (states are images)