AI revision notes

Shaun Toh

August 21, 2019

Contents

1 Week 1 **3**

2 Week 2 **4**
2.1 Basic Pytorch 4
2.2 Logistic Regression 10
2.2.1 Cross entropy loss 11

3 Week 3 **11**
3.1 Neural net basics 11
3.2 One hot vectors 11
3.2.1 Universal Approximation Theorem 12
3.3 Multiclass Classification/Multilabel Classification 12
3.3.1 Multiclass classification 12
3.3.2 Multilabel classification 12
3.4 Overfitting 13
3.5 Better models with less overfitting 13
3.6 Convolutions 14
3.6.1 Convolution output size 14

4 Week 4 **14**
4.1 Finetuning 14
4.2 State of the art lecture(well at that timepoint anyway) 14
4.2.1 Dropout layer 14
4.2.2 Googlenet/Inception's Dimensionality Reduction 15
4.3 Resnets 15
4.4 Batch Normalization 15
4.5 Densenets 16
4.6 Inception V3 16
4.7 BackPropagation 16

5 Week 5 **17**
5.0.1 Problem with gradient propagation . . . 17
5.1 Interpretability . . . 18
5.1.1 t-sne . . . 18
5.1.2 Gradients . . . 19
5.1.3 Lime . . . 20
5.2 Guided Backpropagation . . . 20
5.3 LRP . . . 20

6 Week 6 **21**
6.1 Markov Decision Process . . . 21
6.1.1 Conditional Independence . . . 21
6.1.2 Important descriptors for World states . . . 21
6.1.3 MDP elaboration . . . 21
6.1.4 Partially observable Markov Decision Process . . . 22
6.2 Reinforcement Learning . . . 22
6.2.1 Value function calculations . . . 22
6.2.2 Bellman Equation . . . 23
6.2.3 Computing $V(s)$ . . . 24
6.2.4 Q-function . . . 24
6.2.5 Policy Improvement theorem . . . 25
6.2.6 Q-learning update . . . 25
6.2.7 SARSA . . . 26
6.3 N-step bootstrap . . . 27

7 Week 8 **27**
7.1 Policy Gradient Theorem . . . 27
7.1.1 Usage . . . 28
7.1.2 Usage with Baseline . . . 28

8 Week 9 **28**
8.1 Problem Solving Agents . . . 28
8.2 Tree search Algorithms . . . 29
8.2.1 Breadth First search . . . 29
8.3 Depth first Search . . . 31
8.4 Uniform Cost search . . . 32
8.5 Iterative Deepening Search . . . 33
8.6 Best First Search . . . 34
8.6.1 Greedy search algorithm properties . . . 34
8.7 A* search . . . 34
8.7.1 Heuristics . . . 34
8.8 Problem Relaxation . . . 35
8.9 Local Search Algorithm . . . 35
8.9.1 Hill climbing problem . . . 35
8.9.2 Local Beam Search . . . 35
8.9.3 Genetic Algorithms . . . 35

9 Week 10 **36**
9.1 Rational Agents . . . 36
9.2 Constraint satisfaction Problems . . . 36
9.3 Constraint graphs . . . 37
9.4 Cryptarithmetic . . . 37
9.5 Backtracking Search . . . 38
9.5.1 Improvements . . . 38
9.6 Forward Checking . . . 38
9.6.1 Arc Consistency . . . 38
9.7 CSP - Tree structured or almost . . . 39

10 Week 11 **39**
10.1 Planning and Search . . . 39
10.2 Planning Assumptions . . . 39
10.3 STRIPS . . . 40
10.4 PDDL . . . 40
10.4.1 Domain File . . . 41
10.4.2 Problem Files . . . 41
10.5 Delete Relaxed Problem . . . 41
10.6 H+ heuristic . . . 42
10.7 Reasoning . . . 42
10.7.1 h_{max} . . . 42
10.7.2 h_{add} . . . 42
10.7.3 h_{ff} . . . 42

11 Week 12 **42**
11.1 Uncertainty . . . 42

1 Week 1

- Essentially all neural nets are mapping functions, mostly performing dimensionality reduction on the inputs to produce an "answer".
- The basic idea is the function $f_w(x) = x \cdot w = \sum_{d=1}^{D} x_d w_d$, where $w \in \mathbb{R}^{\mathrm{D}\times 1}$. One can optionally opt to add a bias, making it $x \cdot w + b$
- In order to then optimise your mapping function, one uses Loss, to tell you how to shift the parameters of the gradient to obtain the answer in your training set. One example of loss is mean square error loss, or another example, $\sum_{d\in \mathrm{D}}^{i} Y^i_{predicted} - Y^i_{actual}$, the difference between outputs across all dimensions summed up.
- The idea behind adding a bias is because the bias shifts the decision hyperplane of your output. Consider a simple example where your neural net outputs either 1 or -1 depending on whether the input falls on one side, or on the other. (if it's on the plane itself, output 1).

What the bias does is manually shift the entire decision plane. If the bias is positive, the plane is shifted in the direction of the weights. Otherwise, it is shifted in the opposite direction. (Just unit vector your weights to see the direction).

- The exact solution to your input training set can be solved as follows: Assuming it's $f(x) = w \cdot x$,

$$w = (X^T \cdot X)^{-1} X^T \cdot Y$$

Where the matrix inverse $(X^T \cdot X)^{-1}$ has to exist, and X is the inputs.

- Overfitting happens sometimes due to picking up of noise as opposed to important features within the training data. The weights end up listening to noise, and might end up fitting the training data perfectly but fail at the test data. One way to reduce this possibility is to use ridge regression:

$$argmin_w \sum_{i=1}^{n} (x_i \cdot w_i - y_i)^2 + \lambda ||w||^2$$

Lamda becomes a hyperparameter, and the exact solution is now

$$w = (X^T \cdot X + \lambda I)^{-1} X^T \cdot Y$$

- 0-1 Loss: $L(f(x), y) = 1[sign(g_{w,b}(x)) \neq y]$
- Hinge Loss: $L(f(x), y) = max(0, 1 - g_{w,b}(x)y)$
- Radial Basis function:

$$F(x, x') = exp(\frac{||x - x'||^2}{2\sigma^2})$$

Essentially a similarity measure between two points, since $||x - x'||$ is the mean squared distance. σ is a free parameter.

2 Week 2

2.1 Basic Pytorch

1.
```
torch.empty((2,3)) # Initialises an empty tensor of size 2,3

torch.zeros((5,1)) # initialises a tensor filled with zeroes of size 5,1

torch.ones((5,1)) # initialises a tensor filled with ones of size 5,1

torch.new_full((5,1),10) # initialises a tensor
# filled with 10s of size 5,1
```

3. Numpy to tensor:

```
a = np.array([1,2,3])

torch.tensor(a) # returns a tensor that deep copies a.
#So editing a won't change the values in this.

torch.from_numpy(a) # returns a tensor that shallow copies a.
#Editing a will affect the values in this tensor.
```

4. Tensor to numpy:

```
t = torch.ones((5,1))
t = x.data.numpy() # if used with a gradient. i.e normal pytorch
t = x.numpy() # if torch.no_grad()
# was used during creation of this tensor.
```

5. Devices

```
device=torch.device('cuda')
# use cuda:0 or cuda:1 or so on to specify specific gpus.

xg=x.to(device)
xc=x.to(torch.device('cpu')) #nothing more to say.
```

6. Tensor typing:

```
x=x.type(torch.FloatTensor)
# change to a float tensor for example

x.float() # changes it to a float type tensor too.
#note that there might be unknown nuances.

x=x.type_as(a) # change tensor to a's type.
```

7. torch conditional on tensor

```
res=torch.where(x>5,x,y)
# returns a tensor array of trues and false.
```

8. torch max

```
values, indices=torch.max(input, dim)
# returns a tensor array of maximum values along said dimension.
# returns both the values in one tensor, and the indices in other.
```

9. Matrix multiplication

```
torch.mm(a,b)
```

10. trimming extra dimensions

```
torch.squeeze() #remove all singleton dimensions.
# so a = (1,2,1,5) becomes 2,5.
# if you call a.squeeze() you return a "view" of a
# if you call torch.squeeze(a), you edit a.
# if you do torch.squeeze(a,dim =2)
# you end up with a = (1,2,5)
# unsqueeze works the opposite way.
```

11. Torch Einsum

best seen @

https://stackoverflow.com/questions/55894693/understanding-pytorch-einsum

But it's here too now since i copied it.

```
In [16]: vec
Out[16]: tensor([0, 1, 2, 3])

In [17]: aten
Out[17]:
tensor([[11, 12, 13, 14],
        [21, 22, 23, 24],
        [31, 32, 33, 34],
        [41, 42, 43, 44]])

In [18]: bten
Out[18]:
tensor([[1, 1, 1, 1],
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4]])

1) Matrix multiplication
PyTorch: torch.matmul(aten, bten) ; aten.mm(bten)
NumPy : np.einsum("ij, jk -> ik", arr1, arr2)

In [19]: torch.einsum('ij, jk -> ik', aten, bten)
Out[19]:
tensor([[130, 130, 130, 130],
        [230, 230, 230, 230],
        [330, 330, 330, 330],
        [430, 430, 430, 430]])
```

```
2) Extract elements along the main-diagonal
PyTorch: torch.diag(aten)
NumPy : np.einsum("ii -> i", arr)

In [28]: torch.einsum('ii -> i', aten)
Out[28]: tensor([11, 22, 33, 44])

3) Hadamard product (i.e. element-wise product of two tensors)
PyTorch: aten * bten
NumPy : np.einsum("ij, ij -> ij", arr1, arr2)

In [34]: torch.einsum('ij, ij -> ij', aten, bten)
Out[34]:
tensor([[ 11,  12,  13,  14],
        [ 42,  44,  46,  48],
        [ 93,  96,  99, 102],
        [164, 168, 172, 176]])

4) Element-wise squaring
PyTorch: aten ** 2
NumPy : np.einsum("ij, ij -> ij", arr, arr)

In [37]: torch.einsum('ij, ij -> ij', aten, aten)
Out[37]:
tensor([[ 121,  144,  169,  196],
        [ 441,  484,  529,  576],
        [ 961, 1024, 1089, 1156],
        [1681, 1764, 1849, 1936]])

General: Element-wise nth power can be implemented by repeating the subscrip

# NumPy: np.einsum('ij, ij, ij, ij -> ij', arr, arr, arr, arr)
In [38]: torch.einsum('ij, ij, ij, ij -> ij', aten, aten, aten, aten)
Out[38]:
tensor([[  14641,   20736,   28561,   38416],
        [ 194481,  234256,  279841,  331776],
        [ 923521, 1048576, 1185921, 1336336],
        [2825761, 3111696, 3418801, 3748096]])

5) Trace (i.e. sum of main-diagonal elements)
PyTorch: torch.trace(aten)
NumPy einsum: np.einsum("ii -> ", arr)

In [44]: torch.einsum('ii -> ', aten)
Out[44]: tensor(110)
```

```
6) Matrix transpose
PyTorch: torch.transpose(aten, 1, 0)
NumPy einsum: np.einsum("ij -> ji", arr)

In [58]: torch.einsum('ij -> ji', aten)
Out[58]:
tensor([[11, 21, 31, 41],
        [12, 22, 32, 42],
        [13, 23, 33, 43],
        [14, 24, 34, 44]])

7) Outer Product (of vectors)
PyTorch: torch.ger(vec, vec)
NumPy einsum: np.einsum("i, j -> ij", vec, vec)

In [73]: torch.einsum('i, j -> ij', vec, vec)
Out[73]:
tensor([[0, 0, 0, 0],
        [0, 1, 2, 3],
        [0, 2, 4, 6],
        [0, 3, 6, 9]])

8) Inner Product (of vectors) PyTorch: torch.ger(vec1, vec2)
NumPy einsum: np.einsum("i, i -> ", vec1, vec2)

In [76]: torch.einsum('i, i -> ', vec, vec)
Out[76]: tensor(14)

9) Sum along axis 0
PyTorch: torch.sum(aten, 0)
NumPy einsum: np.einsum("ij -> j", arr)

In [85]: torch.einsum('ij -> j', aten)
Out[85]: tensor([104, 108, 112, 116])

10) Sum along axis 1
PyTorch: torch.sum(aten, 1)
NumPy einsum: np.einsum("ij -> i", arr)

In [86]: torch.einsum('ij -> i', aten)
Out[86]: tensor([ 50,  90, 130, 170])

11) Batch Matrix Multiplication
PyTorch: torch.bmm(batch_ten, batch_ten)
NumPy : np.einsum("bij, bjk -> bik", batch_ten, batch_ten)
```

```
In [90]: batch_ten = torch.stack((aten, bten), dim=0)
In [91]: batch_ten
Out[91]:
tensor([[[11, 12, 13, 14],
         [21, 22, 23, 24],
         [31, 32, 33, 34],
         [41, 42, 43, 44]],

        [[ 1,  1,  1,  1],
         [ 2,  2,  2,  2],
         [ 3,  3,  3,  3],
         [ 4,  4,  4,  4]]])

In [92]: batch_ten.shape
Out[92]: torch.Size([2, 4, 4])

# batch matrix multiply using einsum
In [96]: torch.einsum("bij, bjk -> bik", batch_ten, batch_ten)
Out[96]:
tensor([[[1350, 1400, 1450, 1500],
         [2390, 2480, 2570, 2660],
         [3430, 3560, 3690, 3820],
         [4470, 4640, 4810, 4980]],

        [[  10,   10,   10,   10],
         [  20,   20,   20,   20],
         [  30,   30,   30,   30],
         [  40,   40,   40,   40]]])

12) Sum along axis 2
PyTorch: torch.sum(batch_ten, 2)
NumPy einsum: np.einsum("ijk -> ij", arr3D)

In [99]: torch.einsum("ijk -> ij", batch_ten)
Out[99]:
tensor([[ 50,  90, 130, 170],
        [  4,   8,  12,  16]])

13) Sum all the elements in an nD tensor
PyTorch: torch.sum(batch_ten)
NumPy einsum: np.einsum("ijk -> ", arr3D)

In [101]: torch.einsum("ijk -> ", batch_ten)
Out[101]: tensor(480)

14) Sum over multiple axes (i.e. marginalization)
```

```
PyTorch: torch.sum(arr, dim=(dim0, dim1, dim2, dim3, dim4, dim6, dim7))
NumPy: np.einsum("ijklmnop -> n", nDarr)

# 8D tensor
In [103]: nDten = torch.randn((3,5,4,6,8,2,7,9))
In [104]: nDten.shape
Out[104]: torch.Size([3, 5, 4, 6, 8, 2, 7, 9])

# marginalize out dimension 5 (i.e. "n" here)
In [111]: esum = torch.einsum("ijklmnop -> n", nDten)
In [112]: esum
Out[112]: tensor([  98.6921, -206.0575])

# marginalize out axis 5 (i.e. sum over rest of the axes)
In [113]: tsum = torch.sum(nDten, dim=(0, 1, 2, 3, 4, 6, 7))

In [115]: torch.allclose(tsum, esum)
Out[115]: True

15) Double Dot Products (same as: torch.sum(hadamard-product) cf. 3)
PyTorch: torch.sum(aten * bten)
NumPy : np.einsum("ij, ij -> ", arr1, arr2)

In [120]: torch.einsum("ij, ij -> ", aten, bten)
Out[120]: tensor(1300)
```

Broadcasting:
Smaller tensor gets filled from the left with dimensions of 1 until same dimensionality i.e:

```
#a = (4), b = (1,4)
torch.add(a,b) -> (1,4) since a is filled leftwise.
If a = (1,3), b = (4,1) you can torch.add.
Result is 4,3.
If a = (3), b = (1,4) you can't do broadcast with this...
```

2.2 Logistic Regression

Sigmoid Function:

$$\sigma(a) = \frac{1}{1+e^{-x`}} = \frac{e^x}{e^x+1}$$

Differential of sigmoid = $\sigma(x)(1-\sigma(x))$ Logistic function/Logistic curve:

$$f(x) = \frac{L}{1+e^{-k(x-x_0)}}$$

Where x_0 is the x value of the midpoint, L is the curve's maximum value, k = logistic growth rate/steepness of curve.
The logistic regression model plugs in the output of your model into the function above with $x_0 = 0$ and $k = L = 1$.

$$f(x) = \frac{L}{1 + e^{w \cdot x - b}}$$

the output can be used as the probability for 2 classes, i.e confidence that x belongs to class 1.

2.2.1 Cross entropy loss

$L(p(x_i), y_i)$ is essentially $-log(p_1(x_i))$ if the actual item is of class 1, and $-log(p_2(x_i))$ if the object is class 2. i.e the repective probability of the item. This works because log(1) is 0 (the ideal probability for a classifier if the class is correct is 1. The loss is hence equal to:

$$-y_i log(p_1(x_i)) - (1 - y_i) log(1 - p_1(x_i))$$

Remember that i is the sample number.
Technically this works over the entire training input since we want to argmax over the entire training data.

3 Week 3

3.1 Neural net basics

Whether a neuron fires or how much it fires can be dictated by a function of our choosing. Some examples are:

1. Threshold i.e $g(a) = [a > 0]$
2. Sigmoid i.e $g(a) = \frac{1}{1+e^{-a}}$
3. RELU i.e $g(a) = max(0, a)$
4. Leaky RELU i.e $g(a) = a \times I[a > 0] + a \times \epsilon \times I[a < 0]$ where ϵ is any small number of your choosing

Recurrent neural net - can take own outputs of neurons as inputs (there are cycles within the graph)
Feed forward - the graph is a dag, and goes 1 way. in to out.
Depending on your problem you might use both types though.

3.2 One hot vectors

Essentially indicator vectors. You have (0,0,0,0,1,0,0,0). That means that you're saying one class is present, i.e class number 5.

3.2.1 Universal Approximation Theorem

if we have a continuous function on a m-dimensional hypercube ->$[0,1]^m$ so it moves from 0 to 1 inclusive,
and a non-constant bounded continuous activation function, we can approximate any function $f(\cdot)$ arbtitrarily well. i.e

$$\forall x \in [0,1]^m : |g(x) - (\sum_{i=1} u_i a(w_i x + b_i) + b)| < \epsilon$$

The main problem is how to learn data.
Minor things:

1. one neuron with sufficiently large weights can separate a single hyperplane.
2. if you have several neurons, you have several hyperplane separation sets.
3. adding another layer of neurons on top can help to detect hyperplane intersections in a AND like fashion.

3.3 Multiclass Classification/Multilabel Classification

3.3.1 Multiclass classification

Only one class is present, but I could classify it as multiple possible classes. To first predict across k different classes, we use the softmax function to obtain all the probabilities.

$$p_c(x_i) = \frac{e^{z_c(x_i)}}{\sum_{c'} e^{z_{c'}(x_i)}}$$

Essentially each of the outputs are placed on the top part of that fraction, and you divide it by the sum of all outputs (including itself) at the bottom. Keep in mind that there is NO NEGATIVE on the exponential.
However, the softmax only gives us a way to obtain the predicted probabilities. We still need to perform loss on this. This is almost the same as the two classes case. Cross entropy loss:

$$\sum_{c=1}^{C} -y_{i,c} log(p_c(x_i))$$

or put simply, -ve log of all present ground truth classes.

3.3.2 Multilabel classification

More than one class can be present. To handle this, handle each class as a cross entropy problem. Simple and clean.

3.4 Overfitting

How to know there is overfitting:

1. training loss(on trainset)<Validation loss(on validation set)
2. training score > validation source.

How to prevent it/how it could theoretically be avoided:

1. If you know the generating distribution and optimise for it, you can't overfit. Overfitting only occurs because your training set is finite which does not contain all the information. The law of large numbers/central limit theorem can only help so much.
2. Overfitting could occur due to noise, or a function class that makes poor assumptions on extrapolating the relevant knowledge. How you know it's bad is another question altogether.
3. overfitting in classification can mean prediction of wrong label based on what was seen in the training set only. So it's technically unavoidable.
4. Adding more noise however, will help to reduce overfitting somewhat.
5. Complex classifiers overfit faster due to the fact that less data will be forced to each of the "mini classifiers" -i.e neurons or bins or whatever, and as a result, your chance of wrong predictions/overfit is higher.

3.5 Better models with less overfitting

How to do that:

1. Keep weights small with weight regularization.
   - Ridge regression - add $\lambda||w||_2^2$ to your loss function. where it's your euclidean length of your weights.$\lambda$is hyperparameter
   - LASSO regression - add $\lambda||w||_1^1$ to your loss function. where it's your Manhattan length. $\lambda$is hyperparameter
   - Weight decay - take away a percentage of your current weights at every step. for example, just do $w_{new} = 0.9w_{old}$ crude but effective.
2. add noise to your data. But make sure your added noise is going to distort the actual features. But how do you do that..? look at the data.
3. Use a dropout layer. Technically adds noise to your data, but this time you don't even have to mod the data. Keep in mind google has a patent for this tho.
4. Improve gradient flow, such as using Batch normalisation, residual connections, recurrent neural nets like LSTMs. Keep in mind LSTMs or recurrents can't be parallelized.
5. get even more data.

3.6 Convolutions

Convolutions are skipped. Experiment and think about it. Or just read the notes.

3.6.1 Convolution output size

Just use this formula:

$$\frac{W - K + 2P}{S}$$

W = width or height. Whatever.
K = convolution window size. i.e the length of your convolution square. P = padding size
S = Stride. i.e how many you step.

4 Week 4

4.1 Finetuning

Load weights from another model, and use as much of the present weights as you can for your new model.
Keep in mind if you can't use one layer since it's incompatible, don't bother loading weights for the layers underneath it (until the output). Since those weights are irrelvant due to your differing network design.
You can opt to train only the last layer, or to train all layers.

4.2 State of the art lecture(well at that timepoint anyway)

Only contains relevant points.

4.2.1 Dropout layer

Two versions of dropout:

During standard dropout, you actually edit the output of the neural net itself, triggering or muting neurons accordingly. As a result though, when you attempt to output at test time, neural net outputs must be multiplied by the probability of the neuron being muted, since the neural net is used to the neuron values being at $p \times output$

During inverted dropout, you do the same thing, but multiply the output of all neurons by $1/p$, so that in test time you don't have to do any edits.

Ensemble neural nets essentially result each time you perform this. This is because if you drop out a few neurons, it's like training each of the neurons to be more independent from other neural nets.

4.2.2 Googlenet/Inception's Dimensionality Reduction

Essentially, do 1×1 convolutions on the output of the previous layer. Then, you output to lesser channels than the input. Scaling up works the exact same way. In practice:

- Previous layer - 96 channels output, 56×56output. Results in output player of (96,56,56) [Batch dimension is disregarded.]
- Dimensionality reduction layer - 12 channel output, 56×56 output. In order to do this, do a 1x1 convolution window on the input from the previous layer. The output size from this layer is now (12,56,56)

The simple example above illustrates how it works. As a result of this however, you require even less parameters for your neural net while still capturing the relevant items. Before you dimension reduce, you have
($96\times$convolution window size $\times$ output channel number) parameters to account for this latest input, alongside the bias.

But after throwing in this random layer, you only need ($12\times$ convolution window size $\times$ output channel)

Assuming the output channel and convolution window size is the same for both examples, we first note that your dimension reducing layer has 96×12 new parameters (the convolution window in there is 1x1, so one per channel sounds about right). Your dimension reduction has resulted in ($\frac{96}{12}\times$ convolution window size $\times$ output channel less parameters. Unless your dimension reduction uses that many parameters, it's generally ok to perform dimensionality reduction.

4.3 Resnets

Essentially a neural net with shortcuts across layers. The output between the layer before the "skipped" layers and the output from the skipped layers is concatenated together or added together even, depending. This means the neural net can opt to weigh the importance for the skipped layers. Of note is that batch norm occurs after every convolution.

4.4 Batch Normalization

Requires

1. Mean of the minibatch thus far.
2. Variance of the minibatch thus far.

Also, keep a running mean and running variance of minibatches. Use that in your testing.

The batch is then normalised using the mean and variance, and shifted accordingly. This results in 4 hyper parameters for shifting the batch accordingly. This means that the gradient with respect to inputs matters more on direction as opposed to actual values of the inputs. Done on each input channel to the batchnorm layer.

4.5 Densenets

. Resnet ++.
Connect each layer with other layers of the same block with shortcuts. Means there is a lot of capacity for skipping or weighting between inputs.
1x1 convolutions with Batch norm and ReLU before each layer. Reduces parameters in subsequent 3x3 kernels.

4.6 Inception V3

-Avoid harsh compression of dimensionality of feature maps at the start, avoiding bottle necking. Same idea of dimensionality reduction as above.

4.7 BackPropagation

Because neural nets come with many layers, we need to find a way to adjust every layer according to how it affected and resulted in a loss. Therefore, we will require the chain rule. Remember that the output say of 3 neurons, one after the other, is as follows:

$$(((w_1 \times x + b_1) \times w_2 + b_2) \times w_3 + b_3)$$

where x is the input, w_i is the weight from said neuron, and b_i is the weight from said neuron.
When we complete the output, we end up with the loss, which is a result from all the layers. In order to obtain a layer's own gradient with respect to the loss, we note that we actually called the various neurons as follows:

$$n_3(n_2(n_1(x)))$$

By the chain rule, say if we wanted to find the deepest nested n_1's derivative with respect to the loss, since we have the differential $\frac{\partial L}{\partial n_1}$,

$$\frac{\partial L}{\partial n_1} = \frac{\partial L}{\partial n_3}\frac{\partial n_3}{\partial n_2}\frac{\partial n_2}{\partial n_1}$$

Since we can calculate the loss only with relation to n_3 directly, and then the resultant to n_2 directly and so on. Keep in mind that in some cases you should SUM the derivatives, say if one neuron splits into two paths downstream, the derivative of said neuron is the sum of both derivatives from downstream.

5 Week 5

5.0.1 Problem with gradient propagation

1. Gradient value shrinkage- i.e vanishing gradients. You can see that derivatives up the network are actually products of previous derivatives. If all of them are ¡1, this means that we will end up with very small gradient values as you move up the train.

2. Saturation between activation functions. Since sigmoid is generally used as an activation, sigmoid functions can get saturated, and this can result in values very close to zero, killing off gradients down the entire chain in it's differential $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$

3. Exploding gradient. Like vanishing, but the other way round.

After calculating the gradient, we multiply the gradient by the learning rate and subtract the value from our weights accordingly. However, this introduces yet another problem, that is that the learning rate has to be adjusted depending on the situation. There is sometimes a need to quickly reduce the learning rate to prevent an overstep into NaN territory for the output of your neural network. Some methods for getting a better gradient are listed below:

- Stepwise learning rate decay. you reduce your learning rate by a specified amount before hand using the following formula: $\gamma_{new} = \lambda \times \gamma_{old}$ where λ is your hyperparameter, $0 < \lambda < 1$

- Polynomial learning rate decay. this time you do $\gamma_{new} = \frac{n_{old}}{t_\alpha}$, $\alpha > 0$. Keep in mind we don't use this normally because it shrinks too fast.

- SGD with warm restarts: you do stochastic gradient descent as per normal, but decrease the learning rate according to the following

$$n_{current} = n^i_{min} + (n^{(i)}_{max} - n^i_{min}) \times 0.5 \times (1 + cos(\pi \frac{T_{cur}}{T_i}))$$

  Where n is your learning rate, and you have dictated the maximum and minimum learning rate, and T_{cur} is your current period on this cycle of restarts, and i is your cycle number. You should probably increase the number of epochs by the following factor too: $T_{i+1} = T_i \cdot T_{mult}$

- Weight decay: shrink weights towards zero, as i've stated above before. $w = w \cdot \gamma, \gamma \in (0, 1)$. Note that most tool boxes don't do proper weight decay but actually just do a regularisation constant in the loss.

- Utilise Momentum. Momentum is essentially updating your gradient using the following method:

$$w_{t+1} = w_t - m_{t+1}$$

  where

$$m_{t+1} = \alpha m_t + \lambda_t \times \nabla(Current\,Gradient)$$

This means that you constantly update using the momentum, and momentum is a constant weighted version of the current latest gradient and the past gradients, with a decay on the importance/ weightage of the past gradients (since $\lambda_t \times \nabla(Current\,Gradient)$ means that old weights will eventually converge to zero with enough applications of λ. Of note: $0 < \lambda < 1$

- RMSProp: essentially use exponential moving average of the gradients to update.
  Exponential moving average of gradients:

$$d_t = \alpha d_{t-1} + (1-\alpha)||g_t||^2$$

$$w_{t+1} = w_t - \lambda \frac{g_t}{\sqrt{d_t + \epsilon}}$$

  ϵ is a very small number. α is a hyper parameter, affecting how much you weigh the importance of the previous exponential moving averages in the new one, λ is your learning rate.

- Adam: combine RMS prop with momentum term, do it for each dimension separately.

$$d_t = \alpha_1 d_{t-1} + (1-\alpha_1)(g_t[d])^2$$

  This must be done across each dimension, and α, the hyperparameter here is essentially the importance of past exponential moving averages.

$$m_t = \alpha_2 m_{t-1} + (1-\alpha_2) g_t$$

  This is the momentum calculation, across all gradients. The *alpha* here is different from that that is used in exponential moving averages.

$$w_{t+1} = w_t - \lambda \frac{\frac{m_t}{1-\alpha_1}}{\sqrt{\frac{d_t}{1-\alpha_2} + \epsilon}}$$

  If you want to add weight decay, add the following term: $-\lambda \times K \times w_t$ where K is the weight decay constant, another hyperparameter. Note that the EMA actually smooths out weight decay somewhat though..

5.1 Interpretability

5.1.1 t-sne

A method of visualising the similarity between samples by embedding them all within 2 dimensions.
Given high dimensional data, compute the probability that a data point i will vote for another data point being his neighbour j in your classes. This is centered

about the current point, so the probability summed across all other points being the neighbour of the current point is equal to one.

$$p_{j|i} = \frac{e^{\frac{-||x_i - x_j||^2}{2\sigma^2}}}{\sum_{k:k!=i} e^{\frac{-||x_i - x_k||^2}{2\sigma^2}}}$$

Note that it seems that you are basing it off the distance between the current point and the point you currently consider. It's essentially a **SOFTMAX**.

Symmetrize the equation above, resulting in:

$$p_i j = \frac{p_{j|i} + p_{i|j}}{2N}, p_{ii} = 0$$

Essentially, attempt to make the order of arguments presented above mean nothing. So both nodes should state the same probability that the other is his neighbour when asked about each other. This is to ensure that outliers still contribute to the embedding.

Learn a similar but heavy tailed model where each data point votes for another data point being his neighbour. This is optimised using the Kullback-Leibler Divergence. The heavier tailed probability is to allow for larger distances between the different samples in high dimensions to be reproduced by larger distances in the map.
-note:Kullback Leibler divergence is the measure of difference between two probability distributions.

Toggle your perplexity accordingly (it's a hyper parameter) to get the visualisation you want. But you should keep in mind you're mapping from higher to lower dimension. Loss of some data is inevitable and depending on perplexity, you could create links between points that don't exist.

5.1.2 Gradients

You can attempt to interpret a model by tracing the gradients of the output based off the input image, telling you which pixels are contributing the most to the current classification. This however is not necessarily very effective.

The gradient explains which pixels are most sensitive to change the prediction of the output.

The gradient does not explain which pixels are contributing the most.

5.1.3 Lime

Given a test sample, learn a locally linear approximation about it, and K-Lasso the model you are learning. (K-lasso is actually just L1 regularisation on the dimensions where the weights are highest). You train only about those dimensions to obtain the approximation of the different samples.

A large sampling radius allows the learning of correlations between neighbouring data points, since you now have a clear model that points out the direct parts of the input that matter.

This method is conceptually clear, yet requires training about every point individually and this is highly sensitive to how much you sample about the parameters.

5.2 Guided Backpropagation

Do backprop but zero out incoming gradient if neuron activation or incoming gradient was negative. Essentially, when you backprop, now you only increase gradients of neurons that contribute positively to the prediction.

This method is a heuristic, and has no theoretical underpinning, and proves nothing definitely. Looking only at the signs of gradients and signals gives very clean heatmaps though.

5.3 LRP

Layerwise Relevance Propagation is essentially propagation done backwards:

1. Based off the output, propagate backwards with weights. First begin by calculating the relevance of an output from the neuron, i.e

$$\frac{a_i \times w_{ij}}{\sum_i a_i \times w_{ij}} \times R_{prev}$$

   R_{prev} is the relevance of previous layer. In the case of the layer being the one just before the output, it's $\frac{1}{n}$ where n is number of neurons in that layer.

2. Eventually, you will end up with an output to each of the pixels. Dye the picture's pixels based off the relevance to get an explanation.

There are various different kind of rules that can be applied. Say a neuron of $y = g(\sum_d w_d x_d + b)$ and R_y is the relevance of the output, the beta rule computes the next relevance as:

$$R_{prev} \times (1+\beta)\frac{w_d x_d}{\sum_{d'}(w_{d'}x_{d'})_+ + b_+} - \beta\frac{(w_d x_d)}{\sum_{d'}(w_{d'}x_{d'})_- + b_-}$$

d' means not d. where d is the input number, the + or - is the positive or negative components of the input, beta is a hyper parameter.
Keep in mind that you might have to sum the relevance from several other neurons accordingly to obtain the relevance for this particular neuron in the layer before them.

6 Week 6

6.1 Markov Decision Process

6.1.1 Conditional Independence

sets 1 and 2 are conditionally independent given set 3 if one of the 3 conditions hold.

1. $P(X_{set1}, X_{set2}|X_{set3}) = P(X_{set1}|X_{set3})P(X_{set2}|X_{set3})$
2. $P(X_{set1}|X_{set2}, X_{set3}) = P(X_{set1}|X_{set3})$
3. $P(X_{set2}|X_{set1}, X_{set3}) = P(X_{set2}|X_{set3})$

6.1.2 Important descriptors for World states

- Fully or partially observable world
- single vs multiagent
- deterministic vs stochastic world
- state space structure (discrete or continuous)
- discrete vs continuous time

6.1.3 MDP elaboration

The markov decision process requires the following:

- You must know the probability how the world state will change based off your actions
- You must know the probability/function for the rewards
- Agents must be able to sense the world state in some way.
- Agents must map a state to the relevant action.

Stationary MDP is the one covered in the lecture. $P_t = P$,$R_t = R$. i.e the reward and probabilities don't change.
MDPs are defined by:

- State Space S

- Action space - set of all actions - A
- Transition probabilities from one state to another - P
- Reward function , potentially depending on current state, previous state, action, and more. - R

MDPs with a finite state and action space always have one optimal deterministic policy.

6.1.4 Partially observable Markov Decision Process

- Agent senses observation y_t, and maps it to the relevant action
- Variant: Agent can then either keep an internal state n_t and use the internal state and the external state to map to the relevant action
- Variant: Agent can attempt to deduce the probability distribution over the world states and hence use the probability distribution to deduce their actual current state, y_t. From which the agent picks the appropriate action.

Decentralised Partially observable MDP is the same as this, but with multiple agents.

6.2 Reinforcement Learning

In RL, you don't know $P(s_{t+1}|s,a)$, or in english, the probability distribution of your next state given the current state and action, and you also don't know $R(s,a,s')$, the reward function given the new state and the action and the previous state. There is a need to learn based off observations.
Reinforcement learning has a few important parts to it. Policy dictates what you choose when in a specific state.
Expected future reward is the sum across the different possible actions that have been taken thus far. If the time horizon is to infinity, use a discount factor on all past rewards, i.e $r(s) = E_{.....,a_t\ \pi(a|s_t=s_t),...}[\sum_{t=0}^{\infty}\gamma^t r_t|s_0=s]$
Value function of a policy. - Essentially the gamma discounted expected future reward when starting in state S and continuing according to the policy. Of note is that the value function starts out as an estimate and is updated as you step through the policy to obtain a suitable approximation.

6.2.1 Value function calculations

Where s' is the next state,
If rewards and policies are deterministic, $R = R(s,a,s')$, $a = \pi(s)$

$$V^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))R(s,\pi(s),s') + \gamma \sum_{s'} P(s'|s,\pi(s))E(r_1 + \gamma r_2 + \gamma^2 r_3 + ...)$$

Of note however, is that

$$E(r_1 + \gamma r_2 + \gamma^2 r_3 + ..)$$

is actually equivalent to the Value function's output across the next state. Rewriting what is above, we have:

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s))R(s, \pi(s), s') + \gamma \sum_{s'} P(s'|s, \pi(s))V^{\pi}(s')$$

If they are not deterministic, we calculate their expected value and work from there.

$$V^{\pi}(s) = \sum_{s'} \sum_{a'} \pi(s|a) * P(s'|s, a)R(s, a, s') + \gamma \sum_{s'} \sum_{a} \pi(a|s)P(s'|s, a)V^{\pi}(s')$$

Of note is that we take $\pi(a|s)$ as a set of probabilities too since the policy's outcome is also not deterministic.
As a result of this, we begin the..

6.2.2 Bellman Equation

Both of the equations above:
Deterministic -

$$V^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s))R(s, \pi(s), s') + \gamma \sum_{s'} P(s'|s, \pi(s))V^{\pi}(s')$$

and non-deterministic rewards and policies-

$$V^{\pi}(s) = \sum_{s'} \sum_{a'} \pi(s|a) * P(s'|s, a)R(s, a, s') + \gamma \sum_{s'} \sum_{a} \pi(a|s)P(s'|s, a)V^{\pi}(s')$$

Allow us to calculate the relevant values of a function, given its current state. As a result, we can attempt to use it to calculate the optimal policy:

A policy is optimal if $\forall s V^{\pi}(s) = sup_{\pi} V^{\pi}(s)$. Supremum is chosen here, as supremum is the lowest upper bound within the comparison set. We must keep in mind that the set that you are comparing within is a subset of all possible sets, the set that takes into account the current state, and the action, and the next state. (Reward values are calculated using all 3 values.)
We hence end up with the **Bellman Optimality Criterion**
The value function, given the policy and state will require:

$$V^{\pi *}(s) = max_a \sum_{s'} P(s'|s, a)R(s, a, s') + \gamma \sum_{s'} P(s'|s, a)V^{\pi *}(s')$$

i.e maximise the total expected reward value for any state While the policy itself will need to meet the following:

$$\pi^*(s) = argmax_a \sum_{s'} P(s'|s, a)R(s, a, s') + \gamma \sum_{s'} P(s'|s, a)V^{\pi *}(s')$$

i.e maximise the expected value for any state given the current state.

6.2.3 Computing $V(s)$

Since we cannot possibly obtain the expected value for every state in the value function for an unknown time horizon, we must perform fixed point iteration within a single episode (or encounter, whatever). Beginning with the original

$$V_0(s) = E$$

where E is our estimated original value for that state. (it could be anything really, but the closer it is to the real value, the less it fluctuates and we achieve fake convergence faster) We calculate V_1 as:

$$V_1(s) = max_a \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)V_0(s')$$

We continue this until we obtain "convergence". i.e $max_s|V_{k+1}(s) - V_k(s)| < \delta$. Binder notes that you use max_s so he's looking across each of the states, and looking at their maximum Value function returns for each state. That makes perfect sense actually since the suboptimal value outcomes for each of the state don't matter in deciding your optimal policy. We note that because $0 < \gamma < 1$, the second portion, if iterated sufficiently, converges to a constant value, while the front part of the equation will generally return the same or similar values as you iterate to infinity.
Please note that this is the discrete case. The continuous case uses integrals for proving.

6.2.4 Q-function

Q value is the expected future reward when using an action in a specific state, and then continuing with a policy of π. Remember that $V^\pi(s')$ is the value function which is the expected value of being in a particular state when following the policy.

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)V^\pi(s')$$

This also means that $V^\pi(s) = Q^\pi(s, a = \pi(s))$, or if the action is indefinite, $V^\pi(s) = \sum_a \pi(a|s)Q^\pi(s,a)$.
In simple english, Q = expected reward+$\gamma\times$Q for new states s' averaged with probability to land in s'. As a result, Bellman Equations about the Q function instead of V^π are available:
Deterministic:

$$Q^\pi = \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)V^\pi(s')$$

$$= \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)Q^\pi(s', a' = \pi(s'))$$

Stochastic:

$$Q^{\pi} = \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)V^{\pi}(s')$$

$$= \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')Q^{\pi}(s',a')$$

Since your policy choices are on a probability distribution.
Also note that the optimal policy's value function will be equal to the best action chosen by the Q-function, i.e $V^{\pi *}(s) = max_a Q^{\pi *}(s,a)$, and that the optimal policy itself is just $\pi *(s) = argmax_a Q^{\pi *}(s,a)$ **Similar** to the fact that we don't know the Q values at the start, and we learn by observations, we can iterate over the Q values over each step to slowly converge to their actual values.

$$Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)R(s,a,s') + \gamma \sum_{s'} P(s'|s,a)max_{a'} Q_k(s',a')$$

note that this is **DIFFERENT** from normal value iteration above, since you take the action that gave the maximum q value in your second half of the equation, and do not necessarily pick the best action in the first half.

6.2.5 Policy Improvement theorem

Or braindeadly,
if you have 2 policies, π and π', such that $\forall s V^{\pi}(s) = Q^{\pi}(s,\pi(s)) \leq Q^{\pi}(s,\pi'(s))$, then it must hold that

$$\forall s V^{\pi}(s) = Q^{\pi}(s,\pi(s)) \leq Q^{\pi'}(s,\pi'(s)) = V^{\pi'}(s)$$

6.2.6 Q-learning update

In Q learning, we note that experiences within your learning are not universal. That is, the state transitions are not definite in terms of their actual future reward and final reward. As a result, we don't actually update the entire Q value based off a single experience.

$$Q_{new}(s,a) = (1-\alpha)Q_{old}(s,a) + \alpha(r + \gamma max_{a'} Q_{old}(s',a') - Q_{old}(s,a))$$

Where α is your hyper parameter, between 0 and 1, non-inclusive. To ensure we don't end up stuck in a loop, we add a percentage chance to not follow the policy and pick a random action. This allows for new option exploration. This is of course the discrete state version that we're covering. A continuous state version is different. If continuous, one possibility is using the following loss function after computing your new Q value.

$$L(s,a,r,s') = (r + \gamma max_{a'} Q_w(s',a') - Q_w(s,a))^2$$

r is your reward. s' is newstate and so on...
But this is highly unstable in most gradient descent cases. Apparently (don't

know why or how), it's shown that you should use a few things in optimisation, namely Q_{old}, Q_{new} and Q in the optimization to increase stability. So one way the cool kids are doing it is updating weights as follows:

$$w = argmin_w(r + \gamma max_{a'} Q_{target}(s', a') - Q_w(s, a))^2$$

Q_target is a hyperparameter-esque thing we keep constant for a hyperparameter K episodes. We only update Q_target if the current episode $\%K == 0$ of note is that the equation actually means to attempt to push w where the gamma discounted future rewards of the target policy and the current rewards summed up are approximately equal to the current weight's suggested action given the state.
Another thing is that we require a more stable loss function with smaller gradients for large deviations. Hence, Huber loss:

$$l(y, z) = 0.5(y - z)^2 \times I[|y - z| < 1] + |y - z| \times I[|y - z| \geq 1]$$

Please note that $|y - z|$ is actually the manhattan distance. or to put simply, quadratic loss for small values, linear for large values.
Since the experiences are not complete averages, when we do update the model based off experiences, we should randomly draw from an existing pool of experiences and update accordingly, allowing more common experiences to influence our model more.
Keep in mind in the continuous case, you need to approximate inputs as continuous by sampling about a time point.

6.2.7 SARSA

Sarsa is a method of estimating actual Q-function values. SARSA is an on-policy method, computing the quantities for the current given policy. SARSA has 2 variants, one that evaluates the current policy, and another for policy control (learning the optimal policy $\pi*$)
For policy evaluation, we start by assuming all Q values are 0, and starting from a state, begin evaluating Q values, following the policy's stated actions.

$$Q^{\pi}_{new}(s, a) = Q^{\pi}_{old}(s, a) + (r + \gamma Q^{\pi}_{old}(s', a') - Q^{\pi}_{old}(s, a))$$

This will allow you to compute all Q values.
In the case of wanting to optimise to the optimal policy, you also begin by assuming all Q values are 0. You then choose the next action, based off a new policy that is greedy and designed to maximise Q values at every step.

SARSA can obtain a different result from Q-Learning while using ϵ-greedy policy. This is found in the cliffworld game, where the ϵ-greedy policy has significant risks, and is a lot different from following the optimal policy. A different solution will then result.

6.3 N-step bootstrap

Essentially, because we approximate new Q values using this:

$$Q_{new}(s,a) \approx r + \gamma max_{a'} Q_{old}(s',a')$$

But if you are just starting to learn, the Q_{old} is a very bad estimator. This injects a lot of noise into your update. To compensate however, we could first decompose the above statement, by looking at it's future self. At step 3 for example, you have:

$$Q = reward(step1) + \gamma reward(step2) + \gamma^2 reward(step3) + \gamma^3 \sum_{s''} P(s'''|s'',a'') max_{a'''} Q^{\pi}(s''',a''')$$

The n step bootstrap simply calls for you to update at a later date after you have sufficient approximations, by using one of the ways below:

$$Q_{new}(s_0,a_0) = (1-\alpha)Q_old(s_0,a_0) + \alpha(r_0 + \gamma^1 max_{a1} Q_{old}(s_1,a_1))$$

$$= (1-\alpha)Q_old(s_0,a_0) + \alpha(r_0 + \gamma^1 r_1 + \gamma^2 max_{a2} Q_{old}(s_2,a_2))$$

and so on... this is a basis for n-step algorithms like n-step SARSA. (not covered)

7 Week 8

7.1 Policy Gradient Theorem

Where $J(\theta)$is the value of the policy when starting from a state s_0, and θ is your current model's parameters
The goal is to maximise $J(\theta)$'s returns, i.e $\sum_a \pi(a|s_0,\theta) Q^{\pi\theta}(s_0,a)$.
Considering only the episodic case, i.e termination after a set number of steps.
Policy Gradient methods want to compute the gradient about $J(\theta)$.

$$\nabla_\theta J(\theta) = \sum_{t=0}^{\infty} \sum_{s_t \forall S} P(s_0 \to s^{(t)}|t,\pi) \sum_{a^{(t)}} q_\pi(s^t,a^t) \nabla\pi(a^{(t)}|s^{(t)})$$

Or in other words, the gradient is technically the sum across all possible time steps and all states, where we take the probability of moving into another state given the current timestep and the policy, multiplying that by the sum of all q function returns about the gradient of our current policy about the current parameters(how much each parameter affects the policy).
The policy gradient attempts to abuse the fact that this gradient is directly proportional to

$$\sum_{s_t \forall S} \mu_\pi(s) \sum_a \pi(a|s) q_\pi(s,a) \nabla_\theta(a|s)$$

Where $\mu_\pi(s)$ is:

$$\frac{\sum_{t=0}^{\infty} \sum_{s \in S} P(s_0 \to s|t,\pi)}{\sum_{s'} \sum_{t=0}^{\infty} \sum_{s \in S} P(s_0 \to s'|t,\pi)}$$

7.1.1 Usage

Begin by generating an episode, i.e somewhat following the policy: $a_t \approx \pi(a|s_t)$ For each step compute the estimated future reward, $g_t = \sum_{i=0}^{T-t-1} \gamma^i r_{t+i}$ Then, define a loss that utilises both g_t, and the policy with respect to the current parameters and state. Do this for every step AFTER the episode's conclusion, and backprop. The loss is generally: $\gamma^t g_t \nabla_\theta ln(\pi(a_t|s_t, \theta))$

7.1.2 Usage with Baseline

The point of adding a baseline is to have a comparison, or a goal state to ensure that the model has a "goal state" to work towards. This should only be applied after the model has begun converging somewhat however, meaning that we have to set thresholds to toggle it. However, the training rate will still be fluctuating quite a lot.

8 Week 9

8.1 Problem Solving Agents

Problem solving agents use sequence, goal states, problems and an input of the current world state in order to output the next set of actions to reach their goal states.

1. Offline problems - The state, and everything is known. We know all variables such as the probability of state transitions, current state etc.
2. Online problem - Not all parameters are known. Act as you will and can.

Before we begin, we define a few different possible state spaces for both the solution and defined spaces.

1. State space is abstracted - Real world is absurdly complex, and the state space must be abstracted for problem solving.
2. Abstract State - set of real states
3. Abstract Operator - Complex Combination of real actions
4. Abstract solutions - Set of real paths that are solutions in the real world.

Available problem types include:

1. Single state problems - deterministic transitions, start state at least, is fully observable. Problem is static. The optimum can be easily deduced, and the solution is a sequence.

2. Non-observable - Conformant problems: Initial state is not observable, but later states may be observable. Deterministic in transitions, static problem, discrete states.

3. Non deterministic/Partially observable problem - Contingency Problem. Perceived inputs provide new information about the current state. Solution is a contingency plan or a policy. The agent has to continually search and execute actions accordingly.

4. Unknown stage space - All unknown, the online problem as stated above.

8.2 Tree search Algorithms

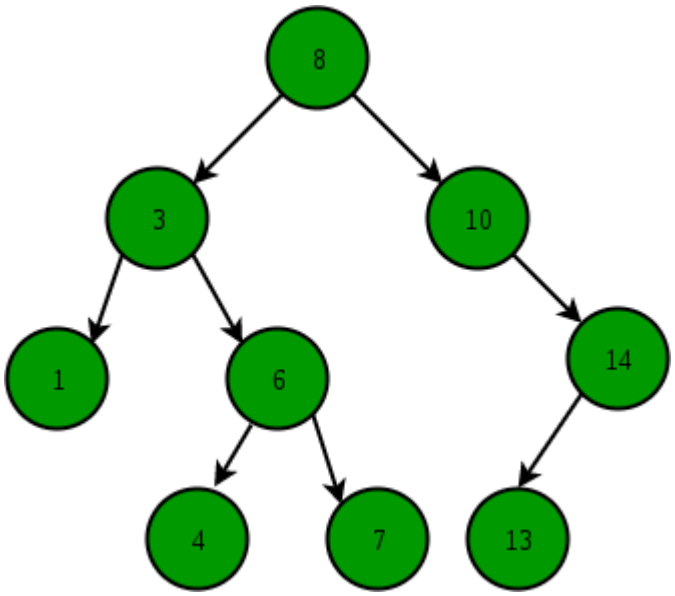
A tree search returns either a solution or a failure. Depending on the search type, it has several defining variables:

- Completeness - Does it always find a solution if one exists?
- Time Complexity - number of nodes generated/expanded
- Space Complexity - Maximum number of nodes in memory
- Optimality - does it always find a least-cost solution

Time and space complexity are measured by maximum branching factor of the search tree. Depth of the least-cost solution, and maximum depth of the state space (up to ∞) This introduces the idea of a fringe, used to hold the queue of nodes that have yet to be considered.

8.2.1 Breadth First search

Fringe is a first in first out queue. Successors go in at the end of the queue. i.e if you have:



We begin the breadth first search with the fringe containing the root node 8. and then...

1. [8]
2. [3, 10]
3. [10, 1, 6]
4. [1, 6, 14]
5. [6, 14]
6. [14, 4, 7]
7. [4, 7, 13]

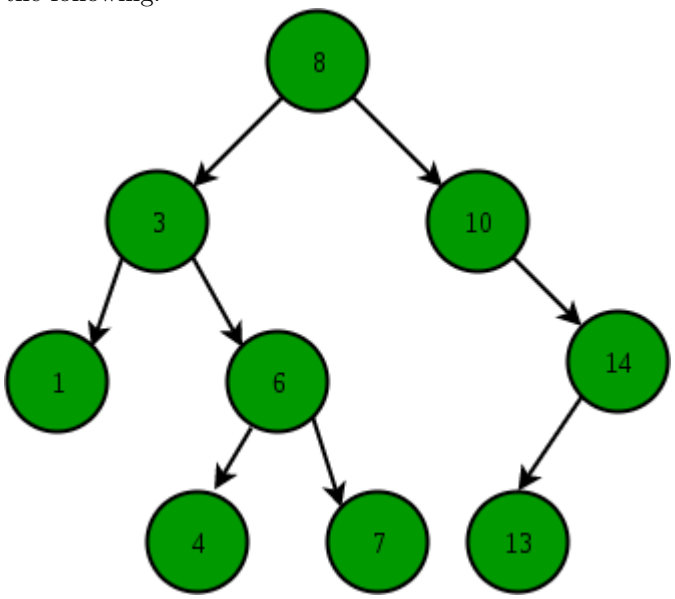
and so on. Should you meet the goal node at any point, TERMINATE.

Breadth First Search - Properties:

- Complete: Yes
- Time Complexity: Exponential in depth of a solution. - $O(b^{d+1})$ where b is the branching factor -i.e max number of possible branches from a single node.
- Space -$O(b^{d+1})$ every node needs to be kept in memory.
- Optimal, search if cost = 1 per step.

8.3 Depth first Search

Similar to depth first, but just propagate downward first. The fringe is a Last in First Out queue. Successors are added to the front of the queue. Again with the following:



1. [8]
2. [3, 10]
3. [1, 6, 10]
4. [6, 10]
5. [4, 7, 10]
6. [7, 10]
7. [10]
8. [14]
9. [13]
10. TERMINATE - out of nodes

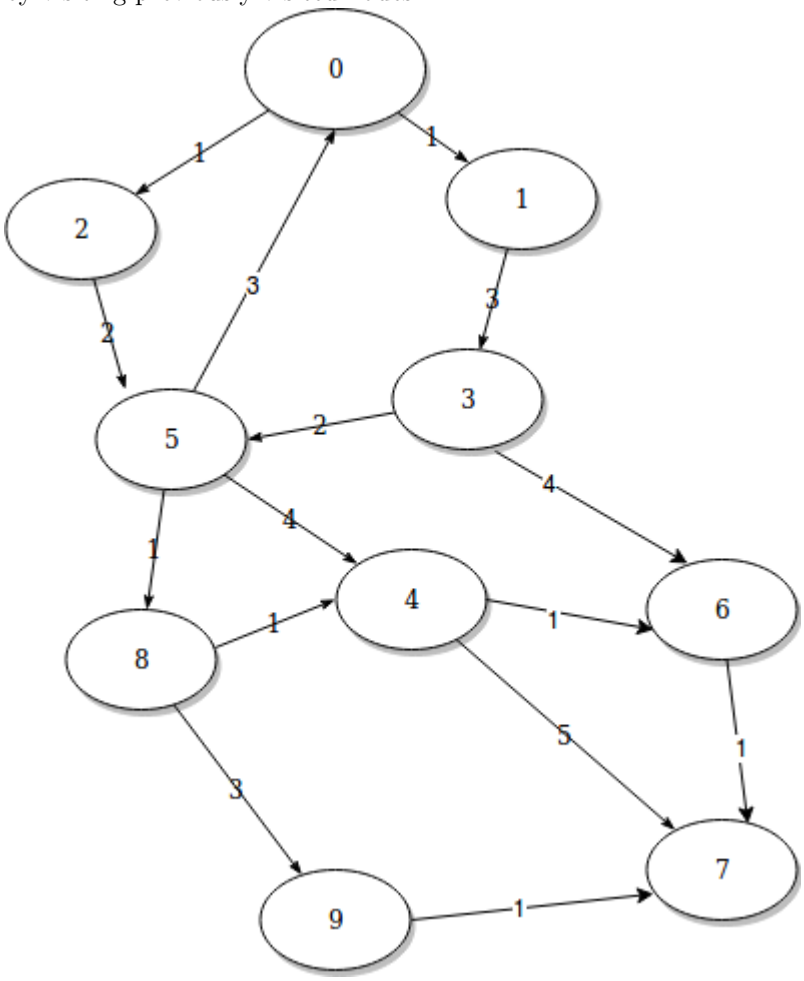
Of course, this is based off the fact that there is no real goal node and we're just expanding till the end. Properties:

- Complete: Yes if the space is finite. i.e doesn't go down the infinite branch.
- Time $O(b^m)$ Where m is the maximum depth

- Space $O(bm)$ i.e linear space
- Optimal - No, unless it happens.

8.4 Uniform Cost search

Expand the current least unexpanded node. The fringe is ordered by increasing cost path (pop the lowest cost path currently). If the cost between all nodes is uniform, it is equivalent to a depth first search. Using the example graph below, where we also note that we will require the not adding of nodes that form loops by visiting previously visited nodes:



Take the goal state as 9, starting from node 0 the fringe is as follows:

1. $[(0, 0)]$
2. $[(02, 1), (01, 1)]$ We ignore the route to 5 as the paths here have direction.

3. $[(01, 1), (025, 3)]$
4. $[(025, 3), (013, 4)]$
5. $[(013, 4), (0258, 4), (0254, 7)]$ We don't add 0 here because we don't want to add loops
6. $[(0258, 4), (0135, 6), (0254, 7), (0136, 8)]$ note how 0135 moves before 0254.
7. $[(02584, 5), (0135, 6), (0254, 7), (02589, 7), (0136, 8)]$
8. $[(025846, 6), (0135, 6), (0254, 7), (02589, 7), (0136, 8), (025847, 10)]$
9. $[(0135, 6), (0258467, 7), (0254, 7), (02589, 7), (0136, 8), (025847, 10)]$
10. $[(01358, 7), (0258467, 7), (0254, 7), (02589, 7), (0136, 8), (025847, 10), (01354, 10)]$
11. $[(013584, 8), (0258467, 7), (0254, 7), (02589, 7), (0136, 8), (013584, 8), (025847, 10), (01354, 10), (013589, 10)]$
12. ... skipping all others up till 02589 since they all terminate at 7 (a dead node),
13. we end with 02589 as the final solution with a cost of 7.

Properties:

- Complete: YES. Always gives a solution if steps cost is positive
- Time - based off number of nodes with a past cost less than that of the optimal solution
- Space - based off number of nodes with a past cost less than that of the optimal solution
- Optimal - Returns the optimal solution with the right checks added to the thing.

8.5 Iterative Deepening Search

Essentially breadth first search with a limit. If you don't find, you can choose to expand the limit. Or in Jun Qing's case, it's a breadth first search, since you always expand anyway.

- Complete: Yes
- Time: $O(b^d)$ branches to the power of depth
- Space: $O(bd)$
- Optimal: Yes, if step cost =1.

8.6 Best First Search

Evaluation Function - Estimates the desirability of each node that can be expanded. Similar to the uniform cost search, you expand the least cost/ most desirable node first.
Use a heuristic as an evaluation function to estimate the cost from the current node to a goal. The best first search is greedy. Expanding this:

8.6.1 Greedy search algorithm properties

- Complete if cycle pruning and finite branches are present, and step costs are lower bounded.
- Time is $O(b^m)$ where b is max number of branches from each node, and m is maximum depth.
- Space is also $O(b^m)$ since all nodes are in memory.
- Generally suboptimal.

8.7 A* search

The general idea is to avoid expand paths that are expensive. Two criterion are taken into account. The first being the cost to reach the current node from the origin, and the second being an estimate of the cost to reach the goal from the current node.
By summing both values, we obtain the estimated total cost of reaching the goal node, with the the confirmed part being the total node cost thus far. In order to make this work, we require a heuristic.
The A* search is always optimal, if the heuristic is consistent.

- A* search is always complete, unless there are infinitely many nodes with the same estimated cost to the goal node.
- Time: Dependent on heuristic. Worst case is exponential in an unbounded space. i.e $O(b^d)$ where d is the depth and b is the branching factor (max branch per node). A good heuristic can avoid this by pruning away many nodes that an uninformed search would expand
- Space: also $O(b^d)$ since all nodes could be stored in memory.
- Optimal solution - Yes, the optimal solution will always be reached. Eventually.

8.7.1 Heuristics

Heuristics are only admissible if they underestimate the cost to reach the node. It must NEVER overestimate the cost to reach the goal node from the current node.

A heuristic is consistent if the its estimate of the cost to the goal from the origin is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour. This means that cost estimates are monotonic. If one cost actually higher than another, the estimate will never estimate it as the lower one.
Heuristic Dominance: if one heuristic is always giving a larger estimate as opposed to another, while both are still admissible, the larger heuristic dominates the other. This is because it is closer to the actual value, while still being lower than the actual cost to the goal.

8.8 Problem Relaxation

A problem with fewer action restrictions is a relaxed problem. Essentially an approximation of the more difficult problem. A solution to the relaxed problem provides information of the original problem.

8.9 Local Search Algorithm

Based off the current state, attempt to improve it. We find a configuration that satisfies the goal state constraints by mutating or making moves. We eventually stop after a certain valued goal has been reached, or a time bound is reached.

8.9.1 Hill climbing problem

"Like climbing a hill that is shrouded in fog" Meant to find the states to a goal state.
Or putting nicely, the simulated annealing search. Allow some bad moves to occur at the start, to escape local minima as you search. However, as time progresses, reduce the probability to eventually converge on the optima. Note that if T decreases slowly enough, the simulated annealing search will find a global optimum with probability approaching 1.

8.9.2 Local Beam Search

For finding an ideal state in the solution space.
Starting with K randomly generated states, search. Ensure all K states are permutated. If any one is a goal state, stop. Else, select the K best successors from the list of all permutations from all states and continue.
But will this always lead to an optima? Not always since you might end up taking them all from one starting point, that led to a local optima that is hard to escape from.

8.9.3 Genetic Algorithms

For finding an ideal state in the solution space.
Successor states are generated by combining 2 parent states. States are represented as a string over a finite alphabet (a string of 0s and 1s). The evaluation

function scores each state, and we produce the next generation of states by selection, crossover, mutation, etc.

9 Week 10

9.1 Rational Agents

Task environment for defining a rational agent (note that i'm only writing this because it's likely for a test. This will only benefit managers who need to bullshit)

1. Performance Measures - Safety, destination, profits, legality, comfort
2. Environment - Expressways, roads, etc.
3. Acutators - Steering, Accelerators etc.
4. Sensor - Video, accelerometer etc.

We also need to define the various kinds of environments:

- Fully Observable vs Partially Observable
- Deterministic vs Stochastic
- Episodic vs Sequential
- Static vs Dynamic environment
- Discrete vs Continuous
- Single Agent vs Multi Agent

9.2 Constraint satisfaction Problems

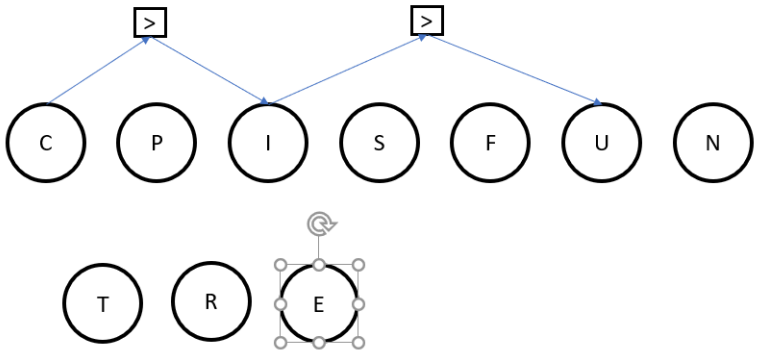
They are defined by their:

- State - variables X_i with values from domain D_i
- Goal test - a set of constraints specifying allowable combinations of values for subsets of variables.

Additional requirements- Complete (every variable is assigned a value), Consistent (no constraint is violated), CSP solution (a complete and consistent assignment to all variables), optionally, Binary CSP. i.e each constraint only relates at most 2 variables.

9.3 Constraint graphs

Nodes are variables, arcs show constraints. nothing more to say.
Depending on the CSP, variables could be discrete with finite domains or have infinite domains, or continuous, solvable by linear programming methods. One example of a constraint graph for the question below, after adding in the constraints that letters above another must be greater than those below for the middle row (C,I,F) as follows:



If you want to solve it, do it yo damn self. The selector on E is left out of spite for the amount of trouble if you had to make the actual graph.
Note that this is a DIRECTED constraint graph. Can constraint graphs be undirected? Yes, depending on whether you even want to add the direction. But generally just listing the condition within the square is sufficient. (Even if it means listing C¿I within the square above.)

9.4 Cryptarithmetic

Cryptarithmetic is just a bunch of possible puzzles. You draw a constraint graph by adding each letter as a node, and try to dictate values that fit in all the constraints between graphs. an Example is..

```
    CP
+   IS
+  FUN
_______
= TRUE
```

one assignment that works is

```
    23
+   74
+  968
_______
= 1065
```

This is of course without any constraints other than fulfilment of the equation. If any other constraints are present, the solution might change.

9.5 Backtracking Search

Consider assignments of values to a single node in the problem when searching for a solution given all constraints. Essentially a depth first search of all possible assignments. Imagine a tree, where every branch from a node, is the permutation of all unassigned nodes and unassigned states from said node. This search is the simplest and most inefficient.

9.5.1 Improvements

Some methods of improvements for backtracking search include:

1. Minimum Remaining Values: the assignment of all nodes higher in the tree with the fewest legal values available for assignment.
2. Degree Heuristic: If a tie occurs from above, choose the variable which assigns the most constraining variables on the remaining variables, greedy searching for a quick resolution.
3. Least Constraining Value: Choose the value with rules out the fewest values in the remaining variables, making it less likely you hit a dead end.

9.6 Forward Checking

Essentially, maintain a record of all possible remaining values for unassigned variables. Terminate the search when any variable runs out of legal values, backtracking till it works. Obviously constraints need to propagate, but some people seem to think it deserves a separate slide on its own.

9.6.1 Arc Consistency

It doesn't actually matter when you propagate the updates for legal values for a nodes, as long as you propagate all of them. You can do propagation after all original constraints are enforced, or upon a domain change. The AC3 algorithm actually requires that all propagations occur immediately. Whether the constraint graph is unidirectional or bidirectional will matter somewhat, but if it's bidirectional, you just need to note when you do constraint propagation. **NOTE:** AC-3 has a complexity of $O(n^2d^3)$, because you potentially need to check all edges between every node, which is n^2, d^2 for checking both domains in each edge, and d because you need to propagate to all remaining domains in the system each time you do the double check of both domains, leading to complexity above.

9.7 CSP - Tree structured or almost

If you have a tree structured CSP, choose a variable as a root node and order variables from root to leaf nodes such that every node's parent precedes it in the ordering. Steps are:

1. Choose a root variable and flatten the graph starting from the root
2. Perform a backward removal phase where we check arc consistency starting from the rightmost node and going backwards
3. Perform a forward assignment phase where we select an element from the domain of each variable going left to right. We are guaranteed that there will be a valid assignment because each arc is arc consistent by way of step 2

Of course, this is assuming you decompose it such that begins to look like a chain.
If the CSP is almost tree structured, we begin by instantiating a variable and pruning its neighbours domains according to the constraints after agreeing with the variable. To extend an iterative algorithm to such CSPs, we allow states with unsatisfied constraints and reassign variable values.

10 Week 11

10.1 Planning and Search

In search, states are represented as a single entity.
In planning, states have structured representations which are used by the planning algorithm.
Planner types

1. Domain Specific - tuned for a specific domain won't work well if at all for other domains.
2. Domain Independent - works in principle for all domains. In practice, needs restrictions on what kind of planning domain.
3. Configurable - domain independent engine, solves based off inputs of domain information

10.2 Planning Assumptions

1. Finite System in terms of actions, states and events.
2. Fully observable state space.
3. Deterministic outcomes for each action.
4. Static - no changes but the controller's actions

5. Goal states are defined.
6. Sequential plans - a plan is a linearly ordered sequence of actions.
7. Implicit time: no time durations, linear sequence of instantaneous states.
8. Off-line planning: planner doesn't know the execution status, plans from current input.

Classical Planning uses simplifying assumptions about the domain and requires all 8 assumptions above.
Since planning reduces to finding a sequence of actions that we take to obtain state transitions, this is just a tree search.

10.3 STRIPS

Stanford Research Institute Problem Solver.

- Originally a planner software.
- Now mainly used to name a formal language to describe planning problems.
- Logic based language that can describe problems, but is limited enough for algorithms to iterate over it.

An instance contains the following: An initial state, the set of goal states, and actions that have preconditions and post conditions attached to them.
Strips is formally defined as a 4-tuple.

1. A set of propositional variables that describe world states.
2. A set of operators i.e actions. Operators have Preconditions, Adds, i.e which facts change to true upon an action being performed, and Del, i.e which facts change to false upon an action being performed.
3. The initial state of the world - true false assignments to the variables from P.
4. The goal states.

10.4 PDDL

PDDLs consist of two things: Domain files and problem files.
Predicates = requirements before we can move something.

10.4.1 Domain File

Domain name indicates the planning problem Domain files look like this:

```
(define (domain gripper-typed)
   (:requirements :typing)
   (:types room ball gripper)
   (:constants left right - gripper)
   (:predicates (at-robby ?r - room)
                (at ?b - ball ?r - room)
                (free ?g - gripper)
                (carry ?o - ball ?g - gripper))

   (:action move
       :parameters  (?from ?to - room)
       :precondition (at-robby ?from)
       :effect (and  (at-robby ?to)
           (not (at-robby ?from))))
)
```

10.4.2 Problem Files

Problem files look like this:

```
(define (problem gripper-x-1)
   (:domain gripper-typed)
   (:requirements :typing)
   (:objects rooma roomb - room
             ball4 ball3 ball2 ball1 - ball)
   (:init (at-robby rooma)
          (free left)
          (free right)
          (at ball4 rooma)
          (at ball3 rooma)
          (at ball2 rooma)
          (at ball1 rooma))
   (:goal (and (at ball4 roomb)
               (at ball3 roomb)
               (at ball2 roomb)
                         (at ball1 roomb))))
```

10.5 Delete Relaxed Problem

Essentially, remove fact negation in all problems. i.e never set anything to false, only things to true are allowed, even in actions. If you can find a solution that

solves the actual problem, you can use the same plan to solve the delete relaxed problem.

10.6 H+ heuristic

The optimal plan for a delete relaxed problem is called the H+ heuristic.
Since the minimal number of steps for a delete relaxed problem $\leq$the total steps for minimal steps to solve a delete-relaxed problem, it will never overestimate the cost of the original problem.
However, h+ is still the computation of optimal plan at every step, and is expensive, therefore we use other heuristics to further approximate it such as h_{add},h_{max},h_{ff}.

10.7 Reasoning

Since Delete relaxed problems can only set more states to true, we just need to find the total number of steps required to set the relevant states to true. For each state, we just do all possible actions until we reach the goal state. This creates a tree for us to search, and we can tell the number of actions required by the depth at which we reach a goal node.

10.7.1 h_{max}

This is the max number of actions needed to achieve the single most costly goal fact. Optimistic Heuristic, since implicit assumption is that an action can set multiple facts to true. However, it can underestimate the true cost by a lot.

10.7.2 h_{add}

Summed cost of all actions for each of the goal states. Pessimistic, assuming that each action can only set one fact to true. Not a admissible heuristic.

10.7.3 h_{ff}

Loop through all goal facts, starting from the last one at a goal state. For each fact, check for all actions that made it possible, continuing all the way till you hit the initial state. Essentially only counting actions that are actively required in achieving the goal state.

11 Week 12

11.1 Uncertainty

Proposition - a whole bunch of atomic events to describe one event:
$(a \vee b) \equiv (\neg a \wedge b) \vee (a \wedge \neg b) \vee (a \wedge b)$
Global Semantics - Express entire distribution as product of local conditional

distributions.
Essentially just sum all events that describe your event.

Conditional Probability- bayes where A = cause B = effect:

$$P(A|B) = \frac{A \wedge B}{P(B)}$$

Product rule for above= $P(A \wedge B) = P(A|B)P(B) = P(B|A)P(A)$
Results in Bayes Rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Inference by enumeration is just summing all the relevant probabilities.

Independence: $P(A|B) = P(A)$ or $P(B|A) = P(B)$ or $P(A, B) = P(A) \times P(B)$

Handling Methods:

- non-monotonic logic - assume it's fine
- Confidence Factor rules
- Probability - use either utility theory(preference based decision making) or probability theory(based off probability $\times$ payoff) or decision theory (both).

For words:

$$P(w_1|c) = \frac{count(w_1, c_i)}{\sum count(w, c_i)}$$

Laplace smoothing - add a small constant alpha to all counts of words.

In a Bayesian network, Total number of required variables = 2^k where k is the number of direct parents for said node.
Conjunctive queries - $P(X_i, X_j|E = E) = P(X_i|E = E)P(X_j|E = E)$