Part II: Deep Learning

What you have learned

The machine learning canon:

- Tools: linear algebra, optimization, sampling, model selection, ...
- Principles: loss, risk, regularization, probabilistic modeling,...
- Algorithms/Problems: classification, dimension reduction, regression,...

All (supervised) methods share a common recipe:

- Frame the problem as learning a function from a family $\mathcal{F} = \{f_\theta : \theta \in \Theta\}$

  $f_\theta : \mathbb{R}^d \to \{0, 1\}$ (or $[0, 1]$) $\quad f_\theta : \mathbb{R}^d \to \Delta_K \quad f_\theta : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2} \quad f_\theta : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$

- Specify a loss function between model and data

  $$L\left(f_\theta(x), y\right) = -y \log f_\theta(x) - (1-y) \log\left(1 - f_\theta(x)\right) \quad L = -\sum_{k=1}^{K} y_k \log f_\theta(x)_k \quad L = \|y - f_\theta(x)\|_2^2 \quad L = \ldots$$

- Minimize the empirical risk on a dataset $\{(x_1, y_1), ..., (x_n, y_n)\}$

$$\theta^* = \text{argmin}_\theta \; \frac{1}{n} \sum_{i=1}^{n} L\left(f_\theta(x_i), y_i\right)$$

Key point: this is machine learning. It works.

But what about all the AI hype?

Modern AI/ML is the same recipe

- Gather data, choose $\mathcal{F} = \{f_\theta : \theta \in \Theta\}$, specify loss, minimize empirical risk
- All the same potential issues exist (wrong $\mathcal{F}$, under/overfitting, optimization issues,...)
- The same statistical and computational thinking is necessary

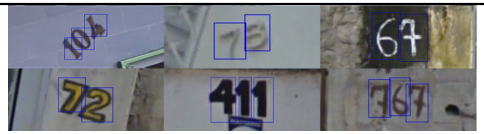
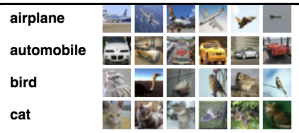
The four catalysts of the AI explosion

1. Large and readily available datasets
2. Massive and cheap computational power
3. Flexible and general function families $\mathcal{F}$
4. Open-source ML software libraries with powerful abstractions

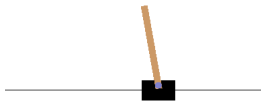
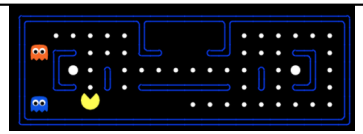
We will study some neural network families $\mathcal{F}$. While neural networks are powerful, there is nothing magical or fundamentally different than what you already know.

CATALYST 1: DATA

Computer Vision

SVHN	CIFAR10	ImageNet	...
			...

Reinforcement Learning

OpenAI Breakout	OpenAI Cartpole	UCB Pacman	...
			...

Natural Language Processing

Wikipedia (English)	Twitter	Jeopardy	...
			...

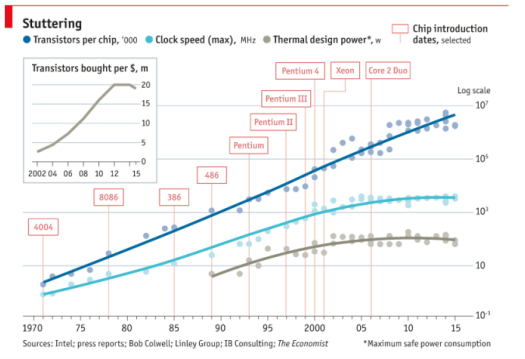
See https://github.com/niderhoff/nlp-datasets

And so much more...

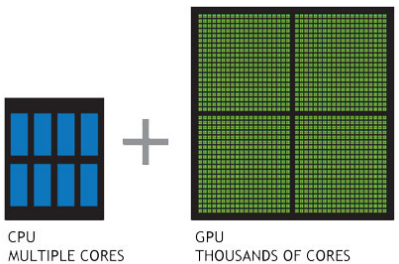
- `https://www.data.gov/`
- `https://opendata.cityofnewyork.us/`
- `https://github.com/caesar0301/awesome-public-datasets`
- `https://data.world`
- ...

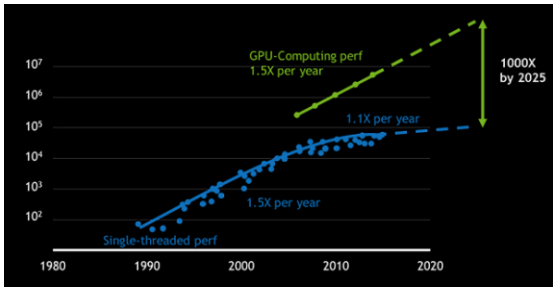
CATALYST 2: COMPUTATIONAL POWER

Processing power has continued to grow... and become cheaper...

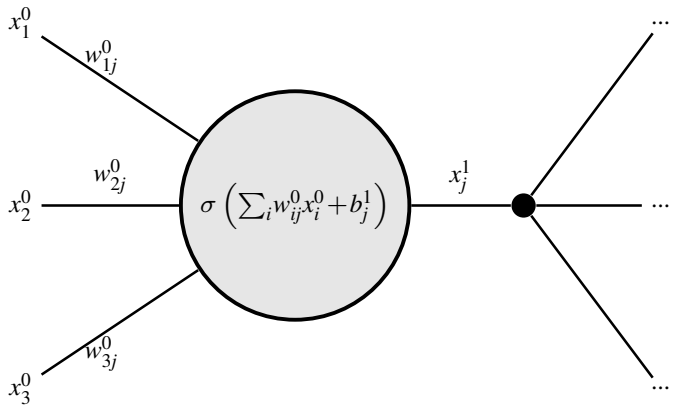


GPUs have accelerated this trend, especially important for ML-relevant computation

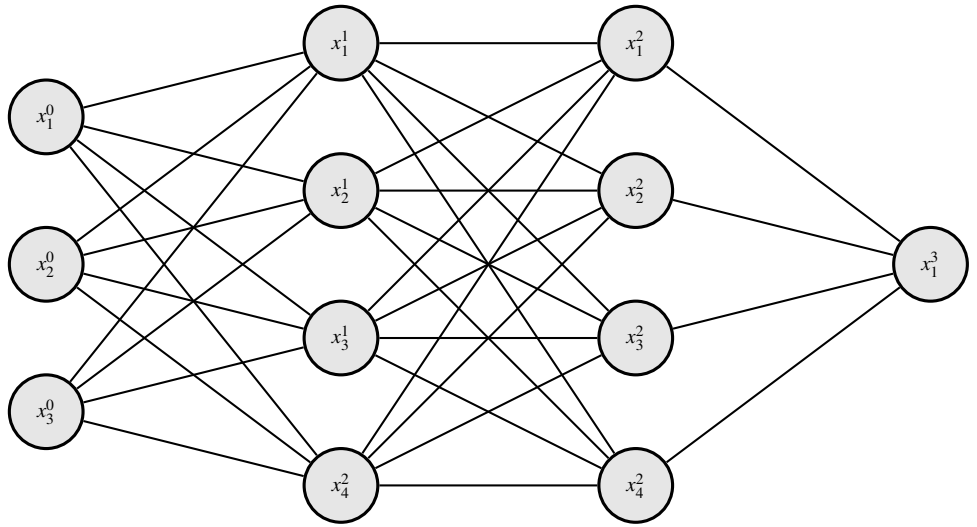




Cloud computing has made this even easier (abstracting away IT and system ops)



Neural unit



With enough layers and enough units per layer, the network is a *universal function approximator*: any function can be fit (given enough data...).

- Inputs x_i^0 enter into unit j, weighted by edges w_{ij}^0, and are summed with bias b_j^1
- $\sigma(\cdot)$ provides elementwise nonlinearity
- The result x_j^1 is transmitted to layer 2, the next layer

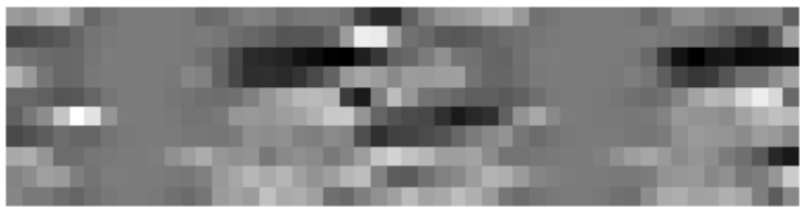
Learning/Training is then minimizing an empirical risk over the parameter set

$$\theta = \left\{ w_{ij}^{\ell}, b_j^{\ell} \right\}_{i,j,\ell} = \{W_{\ell}, b_{\ell}\}_{\ell}$$

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Logistic Regression

x



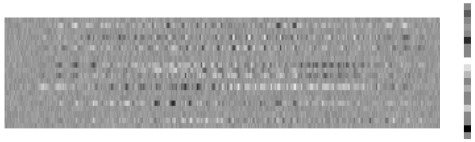
W

b

$f_\theta(x)$
$\sigma(Wx + b)$

Neural Network

x



W_1

b_1

$f_\theta^{(1)}(x)$
$\sigma(W_1 x + b_1)$

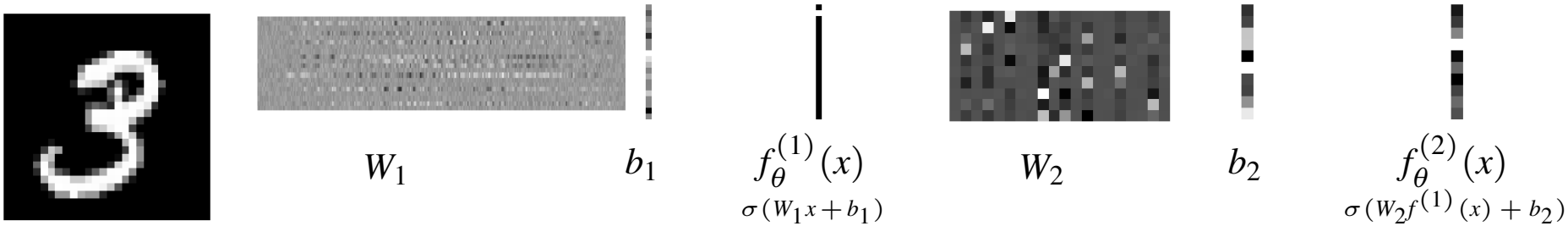
W_2

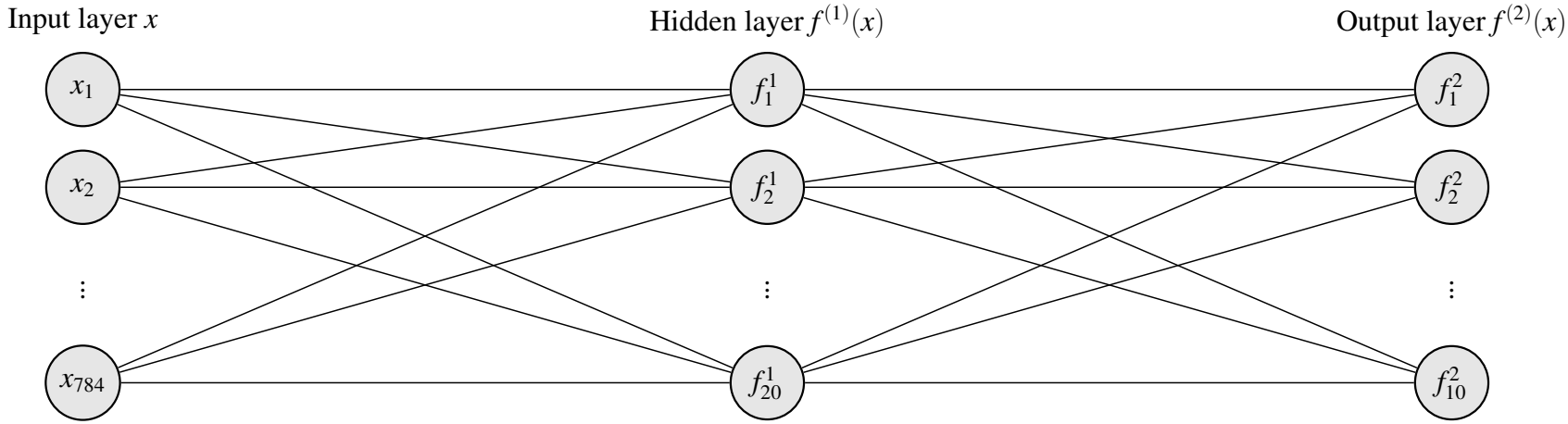
b_2

$f_\theta^{(2)}(x)$
$\sigma(W_2 f^{(1)}(x) + b_2)$

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Neural Network





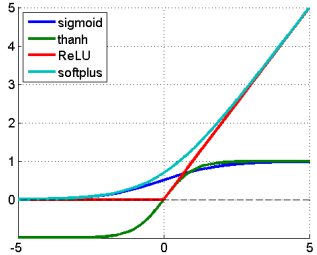
Cascade layers for any amount of depth and complexity!

Naive conclusion: deep learning is easy...

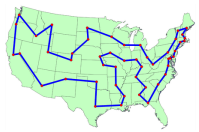
...Deep learning is hard

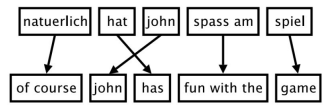
- How do I choose $\left|f^{(1)}\right|$, the number of *units* in the hidden layers?
- How do I choose L, the number of *layers*?
- How do I choose the *activation function* $\sigma(\cdot)$?

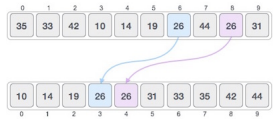
sigmoid	tanh	relu	softplus	softmax	...
$\frac{1}{1+e^{-x}}$	$\frac{e^{x}-e^{-x}}{e^{x}-e^{-x}}$	$\max(0,x)$	$\log\left(1+e^{x}\right)$	$\frac{e^{x_i}}{\sum_k e^{x_k}}$	...



- Are there other choices to make?
- What about overfitting?
- Will my optimizer converge?
- Is my problem solvable with a particular *architecture* $\mathcal{F}$?







- Can my data be fit by a particular *architecture* $\mathcal{F}$?

MNIST vs. SVHN

Deep learning requires engineering skill, statistical thinking, and thoughtful empiricism.

CATALYST 4: SOFTWARE

Machine Learning libraries have abstracted {math, stats, optimization, ...} → engineering

Under the hood are several amazing capabilities. Arguably the two most important:

- Automatic differentiation

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy']
              )
```

- Stochastic optimization

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy']
              )

# Fit the model on training data
model.fit(x_train, y_train, epochs=5,
          callbacks=[tf.keras.callbacks.TensorBoard(log_dir=logs_base_dir, histogram_freq=1)])
```

To understand modern ML, we need to understand why these work... and when they don't.

TOOLS: AUTOMATIC DIFFERENTIATION

REVISITING TENSORFLOW TUTORIAL

Optimization is central to machine learning

- We seek to minimize empirical risk $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f_\theta(x_i)\right)$
- We iteratively optimize to find a point θ^* where $\nabla_\theta R(\theta)|_{\theta^*} = 0$
- Gradient descent (for some *step size* α_k):

$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \nabla_\theta R(\theta)$$

- Note: you will also remember convex optimization and the Hessian H_θ. Neural networks are nonconvex and thus we will largely ignore second order optimization

But no gradients were taken in the tensorflow tutorial!

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['sparse_categorical_accuracy']
              )
```

Somehow tensorflow took the gradients under the hood

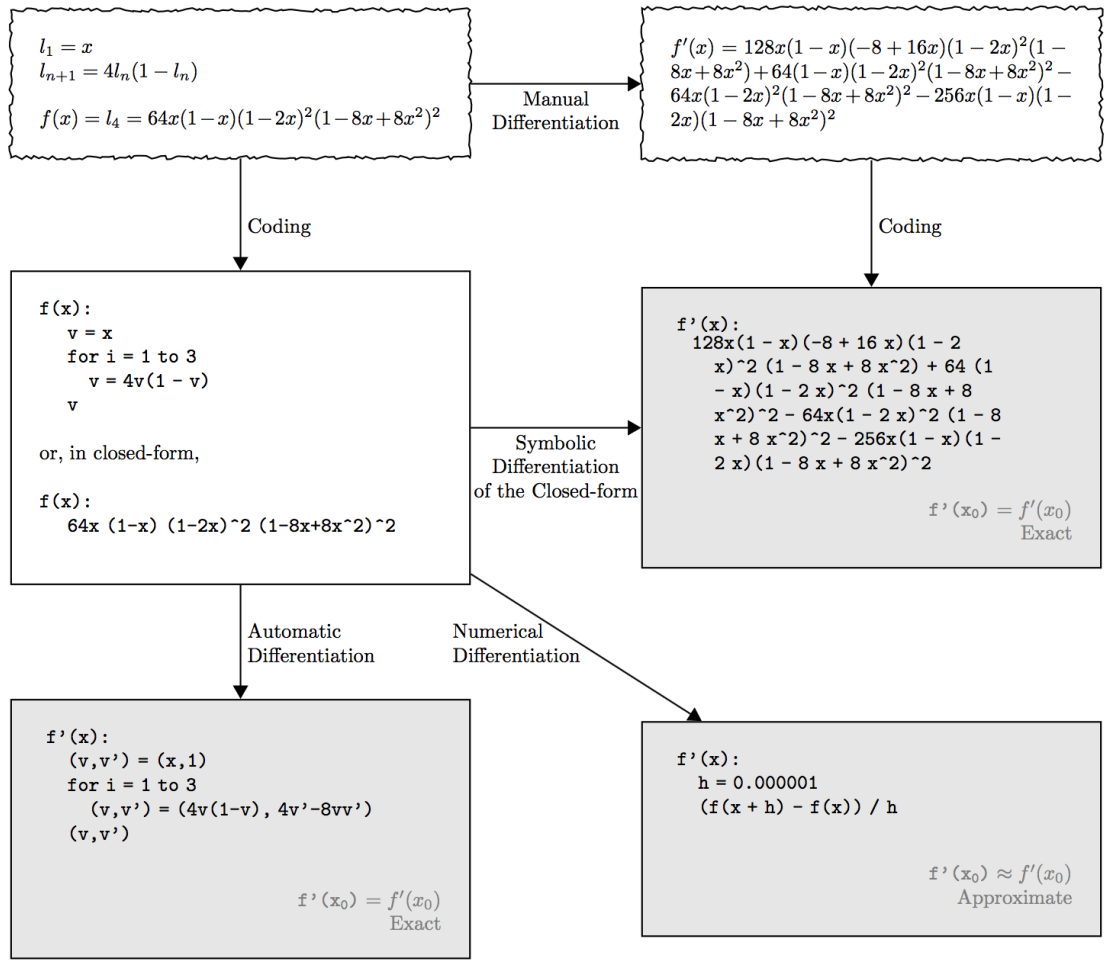
DIFFERENTIATION

Four ways to take derivatives:

- manual (calculus) differentiation
- numerical differentiation
- symbolic differentiation
- automatic differentiation

They are, respectively:

- painful, mistake-prone, not scalable (cost of a Jacobian?)
- unstable (floating point), inaccurate
- restricted (to closed form), unwieldy (expressions)
- awesome: general, exact, particularly well suited to algorithmic code execution



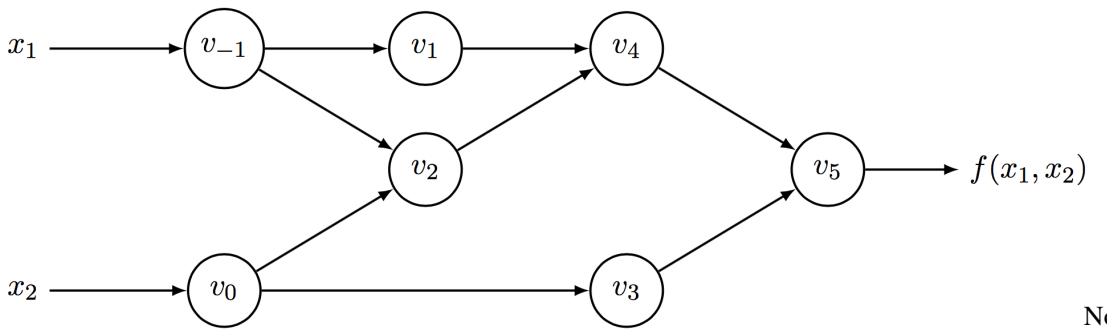
[Baydin et al (2015) JMLR... note the for loop!]

Understanding *autodiff* requires a bit of thinking, but remember, it's just the chain rule

FORWARD MODE AUTOMATIC DIFFERENTIATION

Consider the function $y = f(x_1, x_2) = \log(x_1) + x_1 x_2 - \sin(x_2)$

- Break down f into its *evaluation trace*: $v_{-1} = x_1$, $v_1 = \log v_{-1}$, ...
- List symbolic derivatives for each op in the trace: $\dot{v}_1 = \frac{\dot{v}_{-1}}{v_{-1}}$,...
- Chain rule: recurse through the evaluation trace, numerically calculate (exact!) derivatives



Note: not a neural network.

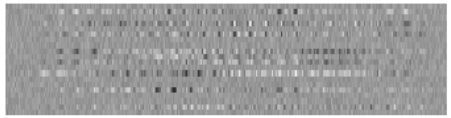
Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= 5.5$

[Baydin et al (2015) JMLR]

Note: it is necessary to execute this forward mode for each input dimension...

REVERSE MODE AND NEURAL NETWORKS

Neural Network



W_1

$f_\theta^{(1)}(x)$

$\sigma\left(W_1 x\right)$

W_2

$f_\theta^{(2)}(x)$

$\sigma\left(W_2 f^{(1)}(x)\right)$

W_3

$f_\theta^{(3)}(x)$

$\sigma\left(W_3 f^{(2)}(x)\right)$

size: $|W_1| = d_0$ d_1 d_2 d_3

$$\left[\frac{\partial}{\partial W_1} L\left(y_i, f_\theta^3(x_i)\right)\right] = \left[\frac{\partial f^1}{\partial W_1}\right] \times \left[\frac{\partial f^2}{\partial f^1}\right] \times \left[\frac{\partial f^3}{\partial f^2}\right] \times \left[\frac{\partial L}{\partial f^3}\right]$$

$d_0 \times d_4$ $d_0 \times d_1$ $d_1 \times d_2$ $d_2 \times d_3$ $d_3 \times d_4$

Computational cost:

- Forward mode: matrix-matrix multiplies $\mathcal{O}(d_0 d_1 d_2 + d_0 d_2 d_3 + d_0 d_3 d_4)$
- *Reverse mode*: matrix-vector multiplies $\mathcal{O}(d_2 d_3 d_4 + d_2 d_1 d_4 + d_1 d_0 d_4)$
- But if L is scalar (like a loss function...), then $d_4 = 1$!

Backprop is reverse mode autodiff on neural network losses. $d_4 = 1 \rightarrow$ very fast and efficient!

NOTES ON AUTOMATIC DIFFERENTIATION

Automatic differentiation is a symbolic/numerical hybrid:

- Each op in the trace supplies its symbolic gradient (e.g., $\dot{v}_1 = \frac{\dot{v}_{-1}}{v_{-1}}$ on earlier slides)
- Execution trace (fwd or bkwd) numerically calculates the exact (not numerical!) gradient

Reverse vs Forward mode autodiff

- Reverse mode is better for $f : \mathbb{R}^N \to \mathbb{R}^M$ for $N \gg M$.
- Forward mode is better for $f : \mathbb{R}^N \to \mathbb{R}^M$ for $N \ll M$.
- What are many machine learning problems? What are (most) neural networks?

Does this only apply to neural nets?

- Most all modern ML libraries include autodiff; hence the computational graph...
- However, not necessary: why not wrap `numpy` ops with their symbolic gradients?

`https://github.com/HIPS/autograd`

Editorial remarks

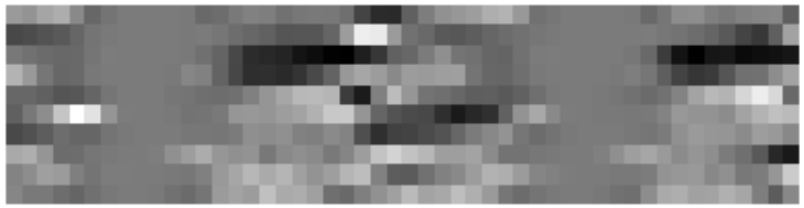
- Audodiff is old and many times reinvented; yes it's just the chain rule.
- Machine learning was embarrassingly slow to adopt autodiff. Now it's pervasive.
- Can I just forget calculus? No! ...but also (sort of) Yes!

TOOLS: STOCHASTIC OPTIMIZATION

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Logistic Regression

x



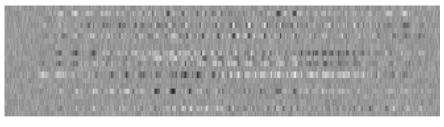
W

b

$f_\theta(x)$
$\sigma(Wx+b)$

Neural Network

x



W_1

b_1

$f_\theta^{(1)}(x)$
$\sigma(W_1x+b_1)$

W_2

b_2

$f_\theta^{(2)}(x)$
$\sigma(W_2f^{(1)}(x)+b_2)$

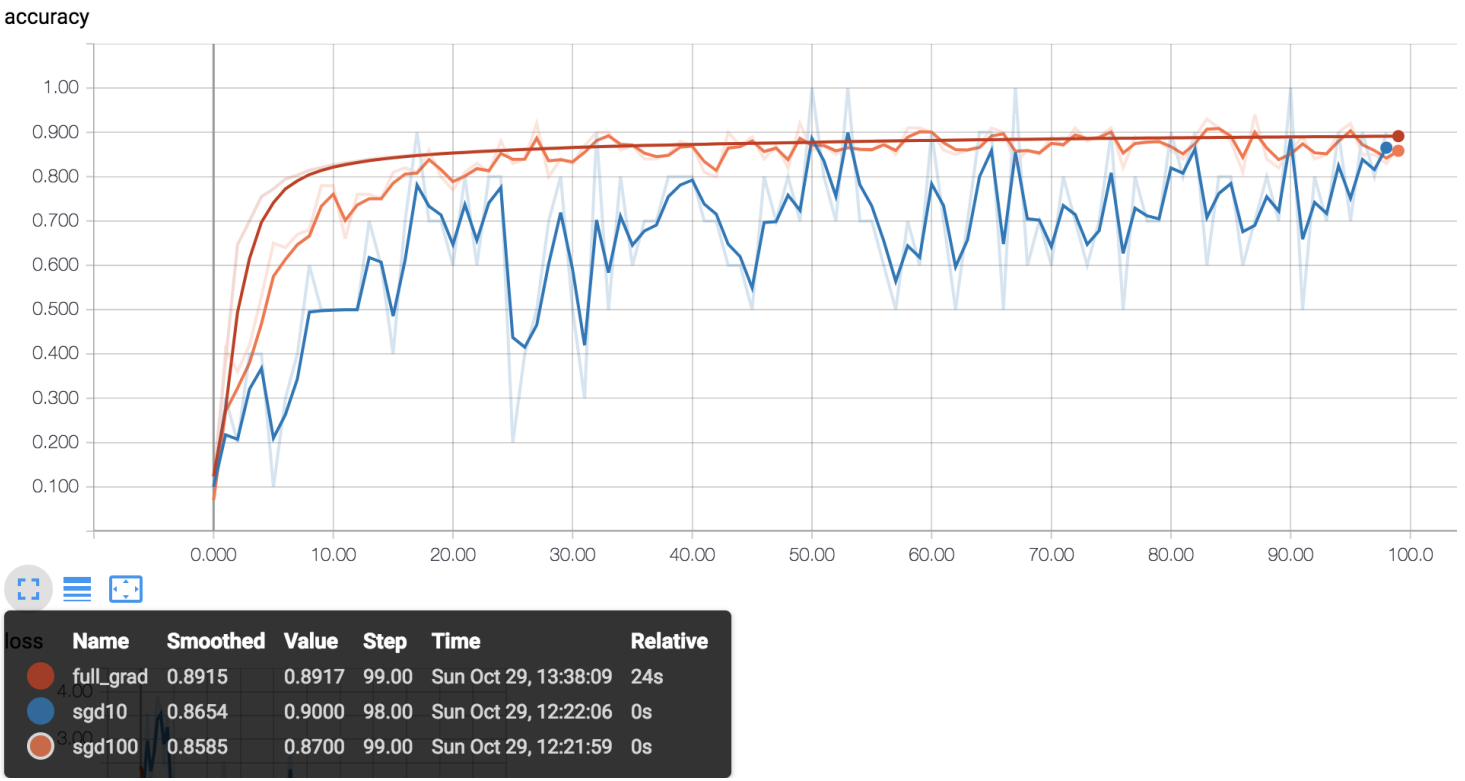
Concerns:

- Number of parameters $|\theta|$ and complexity of optimization is growing...
- With 'big data', at what point will I not be able to reasonably calculate the gradient of the empirical risk $\nabla_\theta R(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta L\left(y_i, f_\theta(x_i)\right)$?
- When will we care about step size α_k in optimization: $\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \nabla_\theta R(\theta)$?

Stochastic Gradient Descent

Idea: at each iteration, subsample *batches* of training data: M random data points $x_{i_1}, \ldots, x_{i_M}$

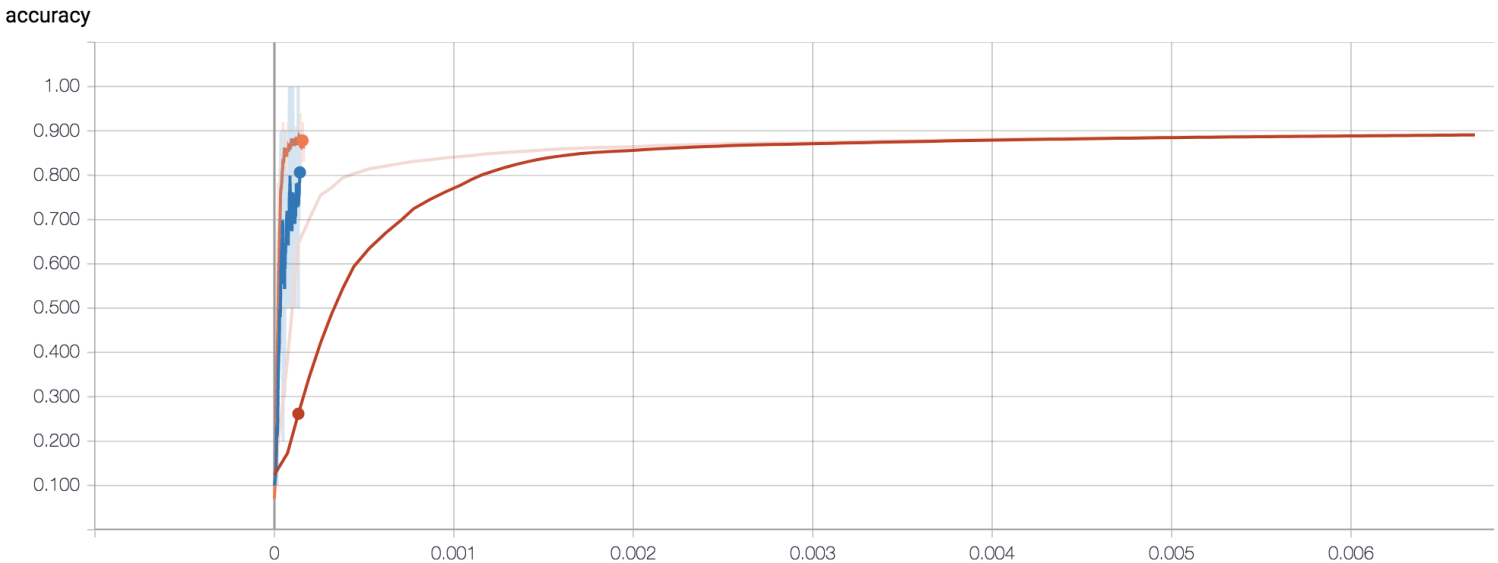
$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \frac{1}{M} \sum_{m=1}^{M} \nabla_\theta L\left(y_{i_m}, f_\theta\left(x_{i_m}\right)\right)$$



Steps are now less likely to be descent directions, hence noisy... but do we gain anything?

Stochastic Gradient Descent

The previous optimization paths, scaled by relative time, show major gains!



Name	Smoothed	Value	Step	Time	Relative
full_grad	0.2617	0.6464	2.000	Sun Oct 29, 13:37:45	0s
sgd10	0.8065	0.9000	98.00	Sun Oct 29, 12:22:06	0s
sgd100	0.8783	0.9200	95.00	Sun Oct 29, 12:21:59	0s

Stochastic Gradient Descent: optimization with noisy (subsampled) gradient estimators

Note: Properly speaking, SGD is batches of size $M = 1$; otherwise *mini-batch* SGD. We will use SGD for both.

Stochastic Gradient Descent

Some common, intuitive, but rather weak arguments that SGD should work:

- Gradients are only locally informative, so needless (early) accuracy is wasteful
- If estimator is unbiased, the stochastic gradient points in the right direction *on average*
- We ideally seek to minimize true risk $E_{p(x,y)}\left(L\left(y, f_\theta(x)\right)\right)$, so already empirical risk $R(\theta) = \frac{1}{n}\sum_{i=1}^{n} L\left(y_i, f_\theta(x_i)\right)$ is a noisy estimator of the true objective
- Injection of noise is likely to kick θ out of saddle points and *sharp* local optima
- Stochastic gradients may help prevent overfitting to the empirical risk function
- Also for discussion: how might batch size help to exploit parallel computation?

The above are roughly correct (or believed so), but careless trust here can be problematic...

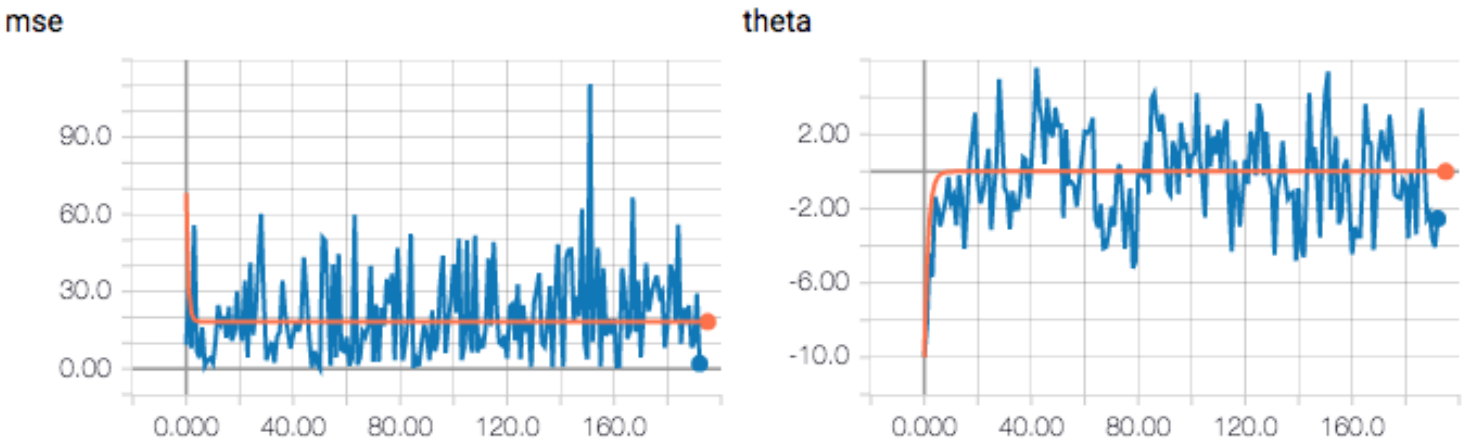
DANGER! SGD REQUIRES CARE

Use SGD to solve this problem:

- Data $\{x_1, \ldots, x_{21}\} = \{-10.0, -9.0, \ldots, 0.0, \ldots, 9.0, 10.0\}$
- Loss $L(x_i, f_\theta(x_i)) = (x_i - \theta)^2$ Note: you should know the answer θ^* already
- Batch size $M = 1$ Note: this choice is just for simplifying the explantion
- Initialize $\theta^0 = -20$
- Step size $\alpha_k = 0.5$ for all k.
- That is, solve:

$$\theta^* = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(x_i, f_\theta(x_i)) = \arg\min_\theta \frac{1}{21} \sum_{i=1}^{21} (x_i - \theta)^2$$

Result: SGD bounces around and never converges...



Takeaway: step sizes $\{\alpha_k\}$ matter tremendously.

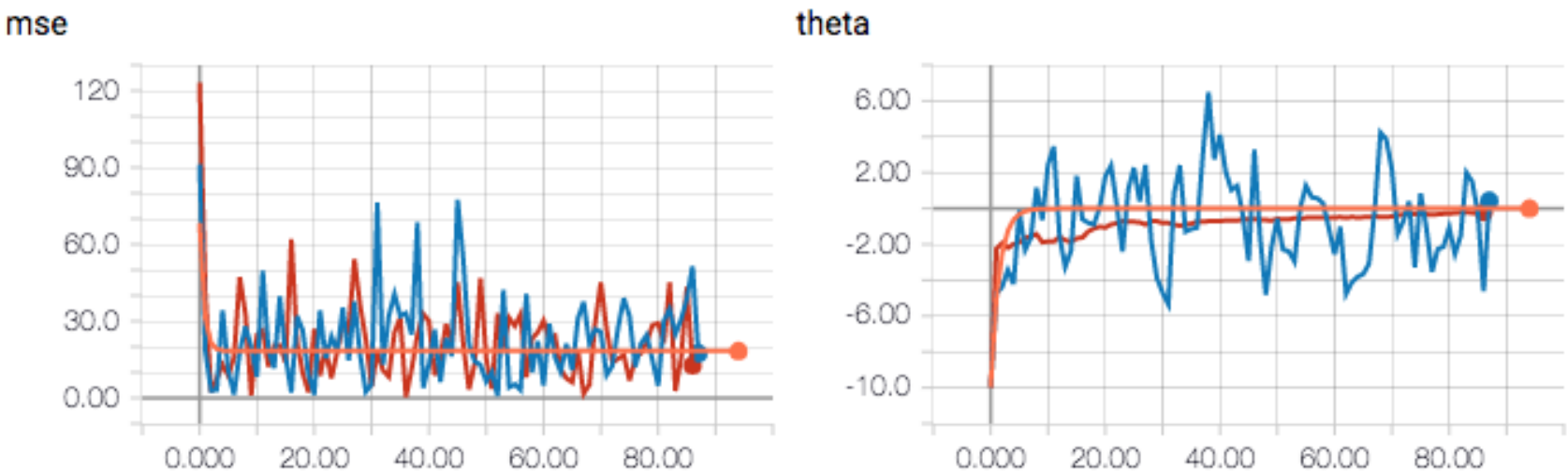
ROBBINS-MONRO

There is a deep literature on SGD. For our purposes:

- Theory: SGD is provably convergent with a proper choice of *schedule* $\{\alpha_k\}_k$
- In brief: Robbins-Monro says $\{\alpha_k\}_k$ must decay quickly, but not too quickly:

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \alpha_k = \infty$$

- A good choice: $\alpha_k = \frac{1}{1+k}\alpha_0$ ...$\alpha_0 = 0.5$ or similar; see `tf.train.inverse_time_decay()`



Orange: full *batch* gradient; Blue: SGD no decay; Red: SGD with decay

SGD is one of the most important enablers of modern machine learning

For those interested, I strongly recommend [Bottou, Curtis, Nocedal 2017] and the original [Robbins and Monro 1951]

More advanced techniques

Can we exploit more information to improve stochastic gradient descent?

- Yes: numerous advances off SGD exist
- No: making rigorous statements about their performance is challenging
- Yes: many cutting-edge methods now use these methods in lieu of standard SGD
- No: there is some indication that they overfit and that SGD is in fact preferred.
- ...an unresolved and very current debate.

Some repeated themes:

- Momentum (Momentum/NAG): $\theta^{(k+1)} \leftarrow \theta^{(k)} - u^{(k)}$ for $u^{(k)} \leftarrow \beta u^{(k-1)} + \alpha_k \nabla_\theta R(\theta)$
- Second order approx. (AdaGrad): $\theta^{(k+1)} \leftarrow \theta^{(k)} - D_k \nabla_\theta R(\theta)$ for a diagonal matrix D_k
- Gradient-based decay (Adadelta/RMSprop/...): $\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \nabla_\theta R(\theta)$ where α_k is a function of previously calculated gradients (such as inverse average squared norm).
- Combinations of above strategies (Adam/...)

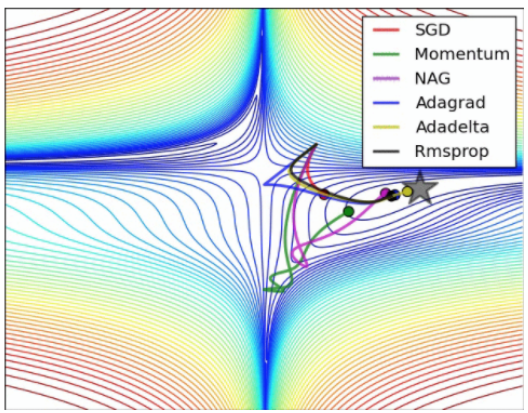


Image 5: SGD optimization on loss surface contours

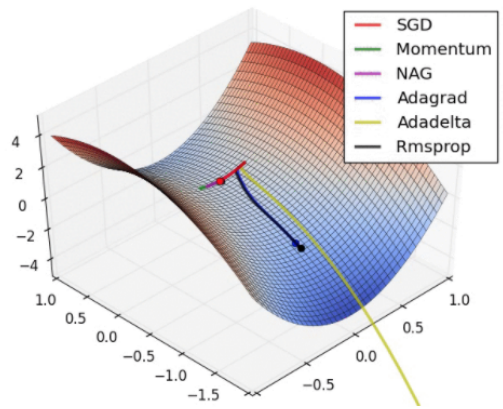


Image 6: SGD optimization on saddle point

image from a blog: `http://ruder.io/optimizing-gradient-descent/`

How to proceed

Practical realities:

- All are implemented in tensorflow, so we allow that abstraction.

  https://www.tensorflow.org/api_guides/python/train#Optimizers

- Try one, tune its hyperparameters, try another, repeat... empiricism matters!
- Current wisdom: use Adam or plain old SGD

For more detail:

- Use SGD, says a leading researcher in this space (Ben Recht)

  https://arxiv.org/pdf/1705.08292.pdf

- A few blogs with heuristic descriptions

  http://ruder.io/optimizing-gradient-descent/

  https://wiseodd.github.io/techblog/2016/06/22/nn-optimization/

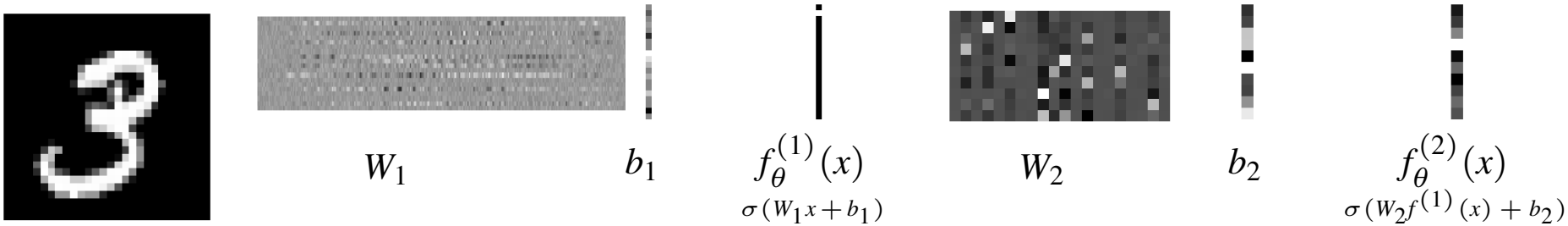
Does this feel abrupt or unsatisfying? It should!

- Choosing step sizes and adaptive gradient techniques are unsolved (nonconvex problems!)
- SGD is rigorous but sometimes slow
- Other methods can be faster but may be problematic in a way we don't yet understand
- Welcome to the cutting edge... this is the "art" (or careful empirical side) of deep learning

CONVOLUTIONAL NEURAL NETWORKS I

INFORMATION BOTTLENECKS IN NEURAL NETWORKS

Neural Network



Notice:

- The first layer bottlenecks the 28×28 space $\mathbb{R}^{784} \to \mathbb{R}^{20}$... loss of expressivity?
- Increasing $20 \to 64$ would drastically increase $|\theta|$... slow algorithm and overfitting!
- ...because every unit sees all input units... that is, W_1 is a *full* matrix

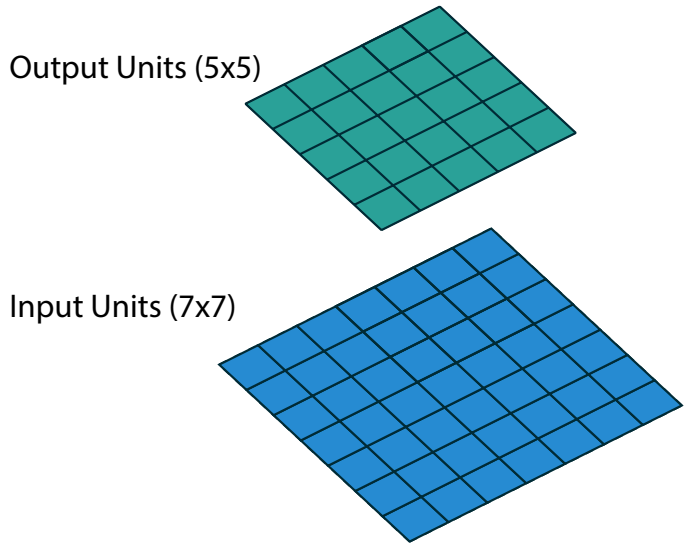
Opportunity:

- What dependency does x_1 have on x_{784}? x_2? x_{29}?
- Recall (from Part I) that exploiting known (in)dependencies is a good thing
- Idea: make linear maps *local*... and rely on later layers to capture long-range features.
- Exploiting *local statistics* allows more outputs for the same net $|\theta|$!

CRITICAL IDEA: LOCAL STATISTICS

A new view of the same *fully connected* layer that we have been using:

- Blue: input units (eg 7×7 image)
- Green: output units (5×5 readout)
- Weight matrix (not shown): $\mathbb{R}^{49 \times 25} \rightarrow |\theta| = 1225$



Local linear *filter*: consider only a 3×3 linear map, and sweep it locally

- New weight matrix: $\mathbb{R}^{3 \times 3} \rightarrow |\theta| = 9$
- $> 400\times$ savings in parameters!
- But we have lost expressivity...

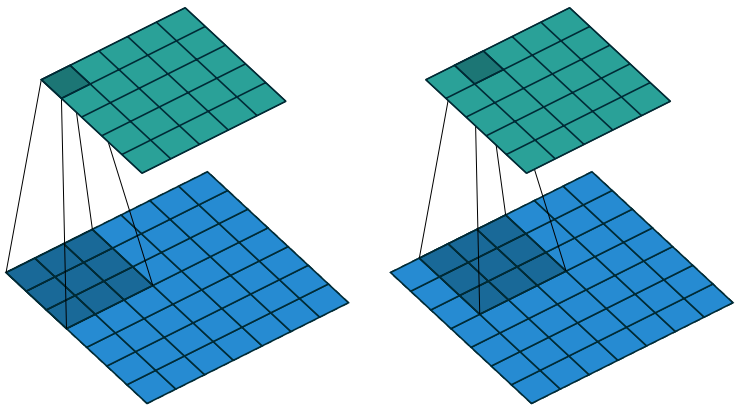
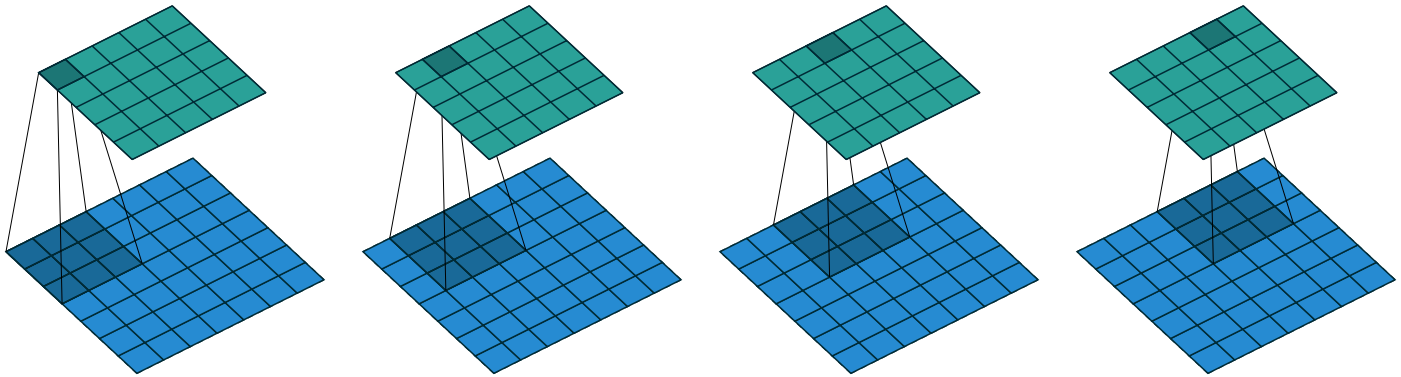


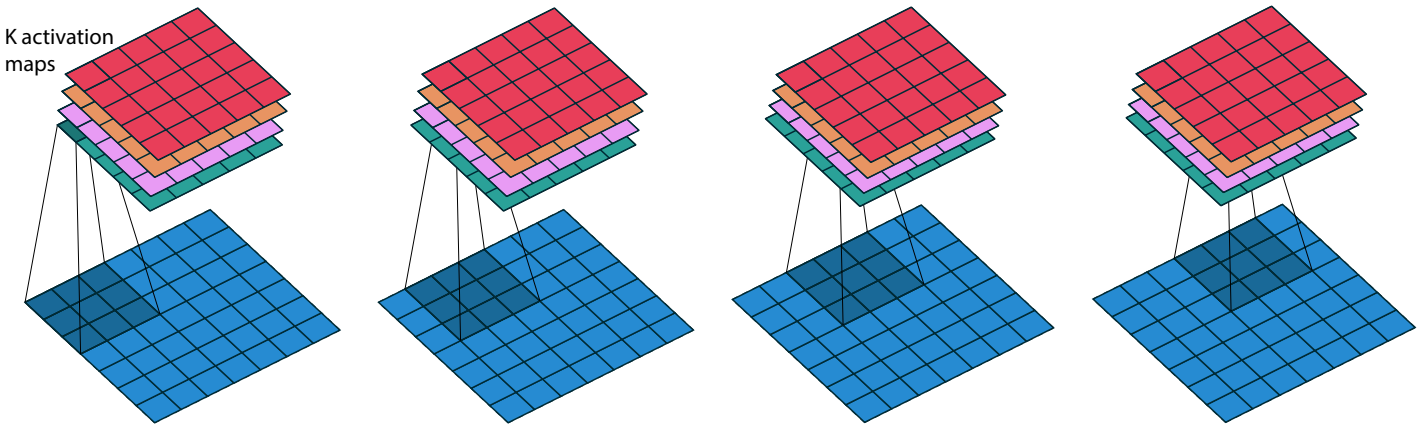
Image credit for all of these and the following: `https://github.com/vdumoulin/conv_arithmetic`

CONVOLUTIONAL LAYER

Call this 3×3 linear map a *filter* or *convolution*



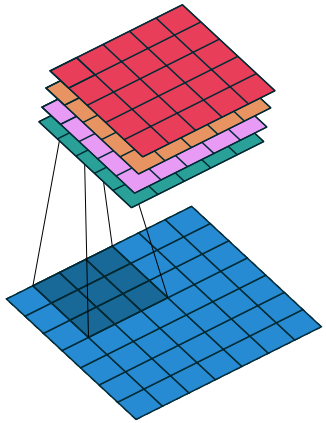
Now use multiple filters (below $K = 4$), producing multiple *activation maps* (each 5×5)



Convolutional layer: linear map applied as above; a $3 \times 3 \times 1 \times 4$ parameter tensor.

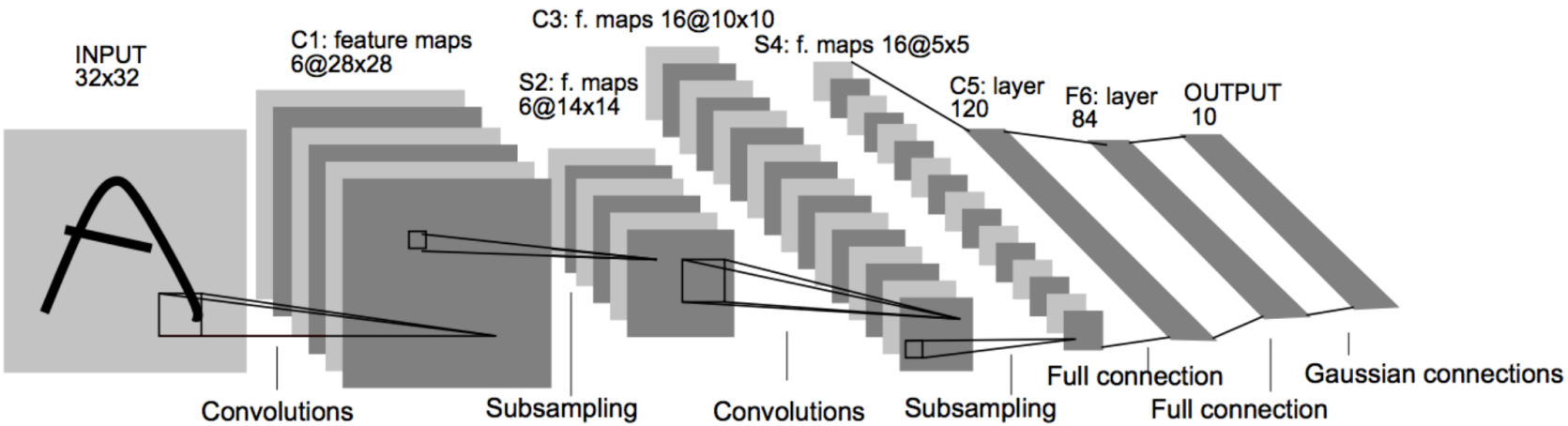
Our/tf convention for 2D convolution: filter width $\times$ filter height $\times$ input depth $\times$ output depth.

Convolutional Neural Network



Convolutional Neural Network: a neural network with some number of convolutional layers. The workhorse of modern computer vision.

You should now be able to interpret/implement published models such as:



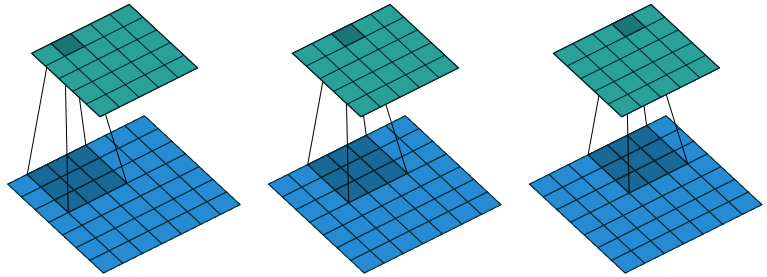
[LeCun et al 1998]

- What is the filter size from input to C1? 5×5
- What is the size of the weight matrix from S4 to C5? $16 \times 5 \times 5 \times 120 = 48,000$
- What is subsampling? It's now called average pooling. What's average pooling? ...

TRICKS OF THE TRADE: ZERO PADDING

Note a few potential drawbacks:

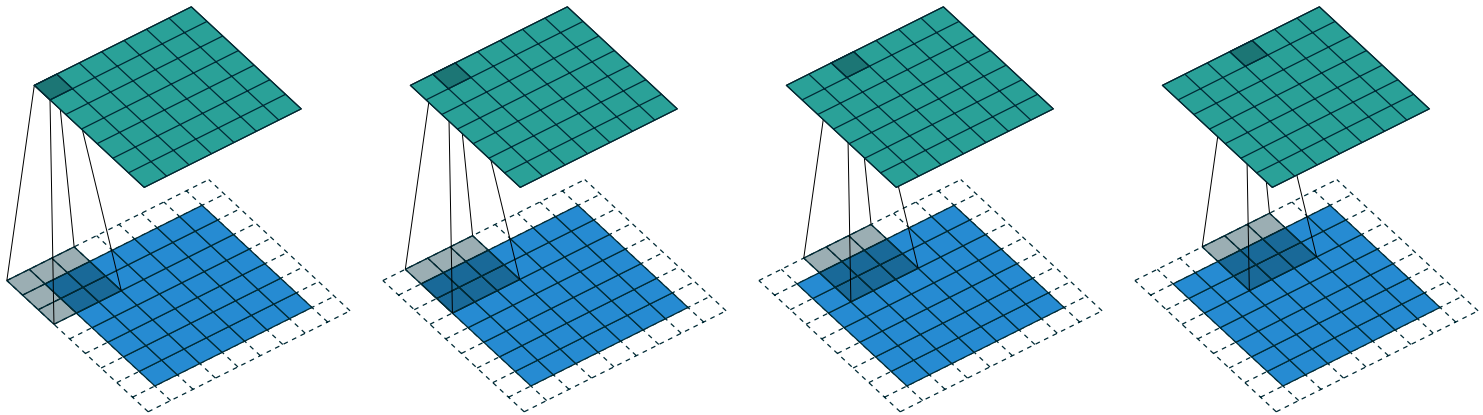
- Filtering reduces spatial extent of activation map
- Edge pixels/activations are less frequently seen
- (Note these can also be benefits)



Zero Padding:

- Add rows/cols of zeros to the input map, solving both problems
- Output activation maps will preserve size when

$$M_{pad} = \frac{1}{2}(M_{filter} - 1)$$

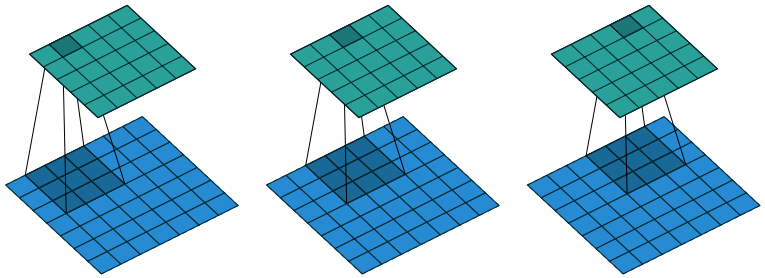


Note: one can zero-pad more/less/asymetrically/otherwise, with varied problem-specific effects

TRICKS OF THE TRADE: STRIDING

On the other hand:

- Filtering processes the same information repeatedly
- Possibly wasteful if images are quite smooth
- Could get more activation maps if each was smaller

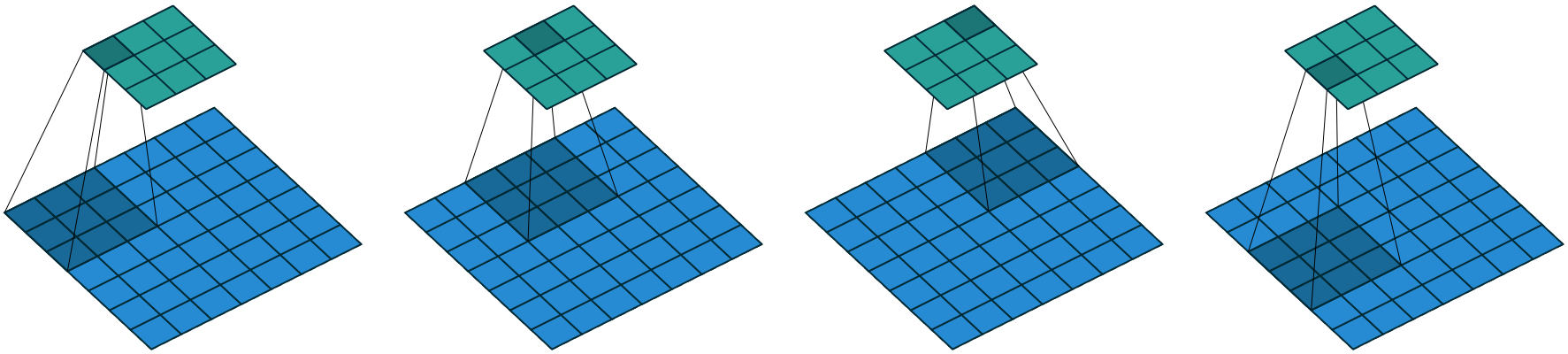


Stride:

- Jump the filter by some M_{stride} pixels/activations
- Output activation map (assuming square) will be of height/width

$$M_{output} = \frac{M_{input} - M_{filter} + 2M_{pad}}{M_{stride}} + 1$$

- Caution! Non-integer results in above will be problematic. Care is required.

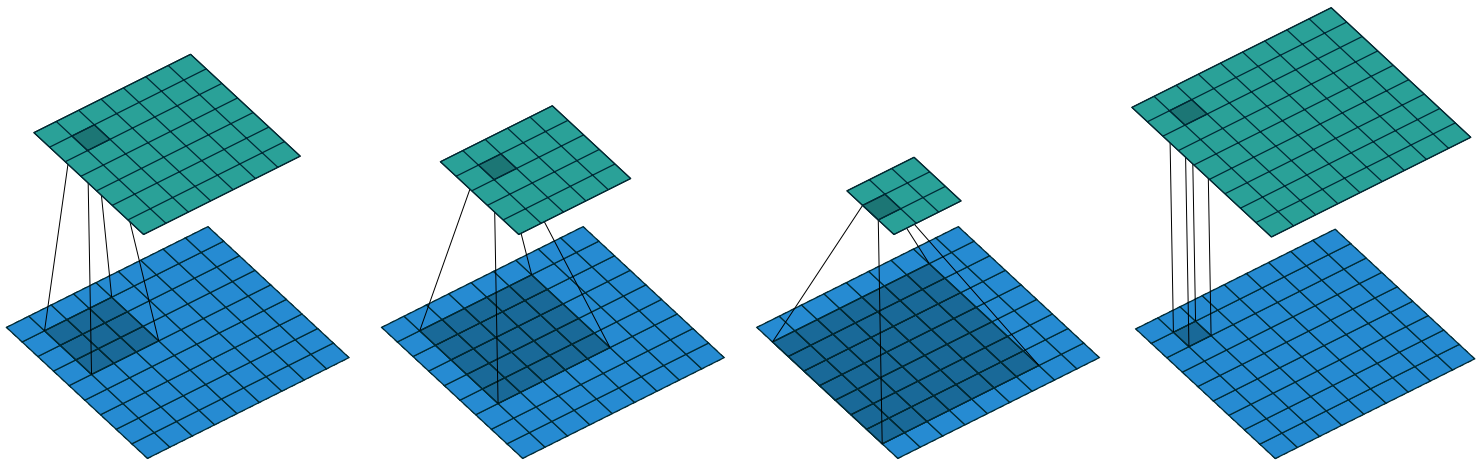


Note: striding and zero-padding give design flexibility and balance each other

TRICKS OF THE TRADE: FILTER SIZE

Notice:

- Smaller filters process finer features
- Larger filters process broader features
- Common choices: 3×3, 5×5, 7×7, 1×1
- Empiricism dictates which to use (again: the art of deep learning)



Wait! What is a 1×1 layer? Isn't that meaningless?

- No! Remember, the conv layer is filter width $\times$ filter height $\times$ input depth $\times$ output depth
- Critical: **filters always operate on the whole depth of the input activation stack**
- 1×1 conv layers $\rightarrow$ dimension reduction: preserve map size, reduce output dimension K

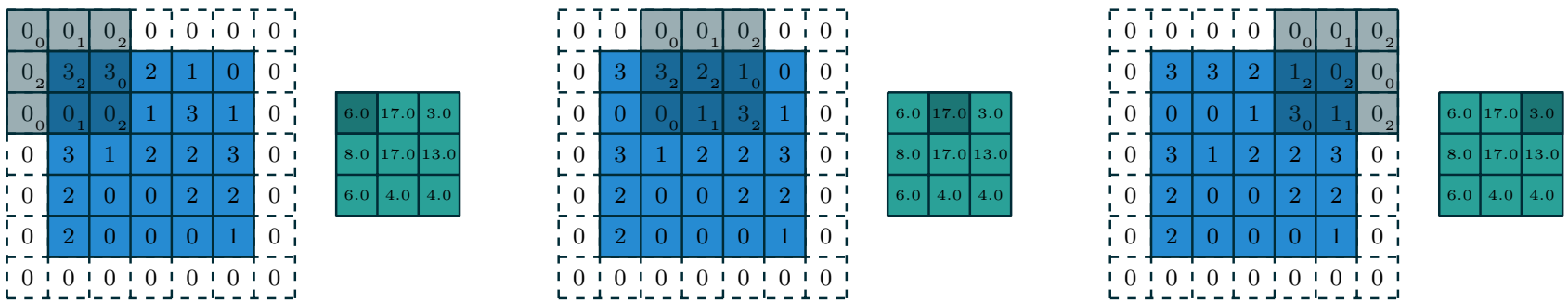
PUTTING THESE ALL TOGETHER

Context

- Convolutional layers specify the linear map (and how to calculate it)
- An elementwise nonlinearity is still expected to follow
- `tf.nn.relu( tf.nn.conv2d( x , W_cnn , strides=[1,2,2,1] , padding='SAME' ) + b )`
- Compare to `tf.nn.relu( tf.matmul( x , W) + b)`

Note: `tf 'SAME'` chooses zero padding to satisfy $M_{out} = \left\lceil \frac{M_{in}}{M_{stride}} \right\rceil$, for stride $= [batch, width, height, depth]$

Specific example



Questions

- What is the filter?
- What is the filter width?
- What is the zero padding?
- What is the stride?

In Practice

Make `cnn_cf`: a single convolutional layer network with 64 activation maps

In [15]:
```
# elaborate the compute_logits code to include a variety of models
def compute_logits(x, model_type, pkeep):
    """Compute the logits of the model"""
    if model_type=='lr':
        W = tf.get_variable('W', shape=[28*28, 10])
        b = tf.get_variable('b', shape=[10])
        logits = tf.add(tf.matmul(x, W), b, name='logits_lr')
    elif model_type=='cnn_cf':
        # try a 1 layer cnn
        n1 = 64
        x_image = tf.reshape(x, [-1,28,28,1]) # batch, then width, height, channels
        # cnn layer 1
        W_conv1 = tf.get_variable('W_conv1', shape=[5, 5, 1, n1])
        b_conv1 = tf.get_variable('b_conv1', shape=[n1])
        h_conv1 = tf.nn.relu(tf.add(conv(x_image, W_conv1), b_conv1))
        # fc layer to logits
        h_conv1_flat = tf.reshape(h_conv1, [-1, 28*28*n1])
        W_fc1 = tf.get_variable('W_fc1', shape=[28*28*n1, 10])
        b_fc1 = tf.get_variable('b_fc1', shape=[10])
        logits = tf.add(tf.matmul(h_conv1_flat, W_fc1), b_fc1, name='logits_cnn1')
```

Note:

- This network should be more expressive than logistic regression
- Compare $|\theta|$ with logistic regression
- Draw this network
- Run it...

CAUTION: NUMERICAL INSTABILITY

Warning

- The `softmax` operation should > 0, but numerically can sometimes be $== 0$
- $\log 0$ will cause your training to crash with some `NaN` errors (possibly just in `tb`)
- Numerical stability is always a concern in practical machine learning
- Here the problem is readily spotted...

```
In [ ]: def compute_cross_entropy(logits, y):
            y_pred = tf.nn.softmax(logits, name='y_pred') # the predicted probability for each example.
            cross_ent = tf.reduce_mean(-tf.reduce_sum(y * tf.log(y_pred), reduction_indices=[1]))
```

- Always use:
  - `tf.nn.softmax_cross_entropy_with_logits`

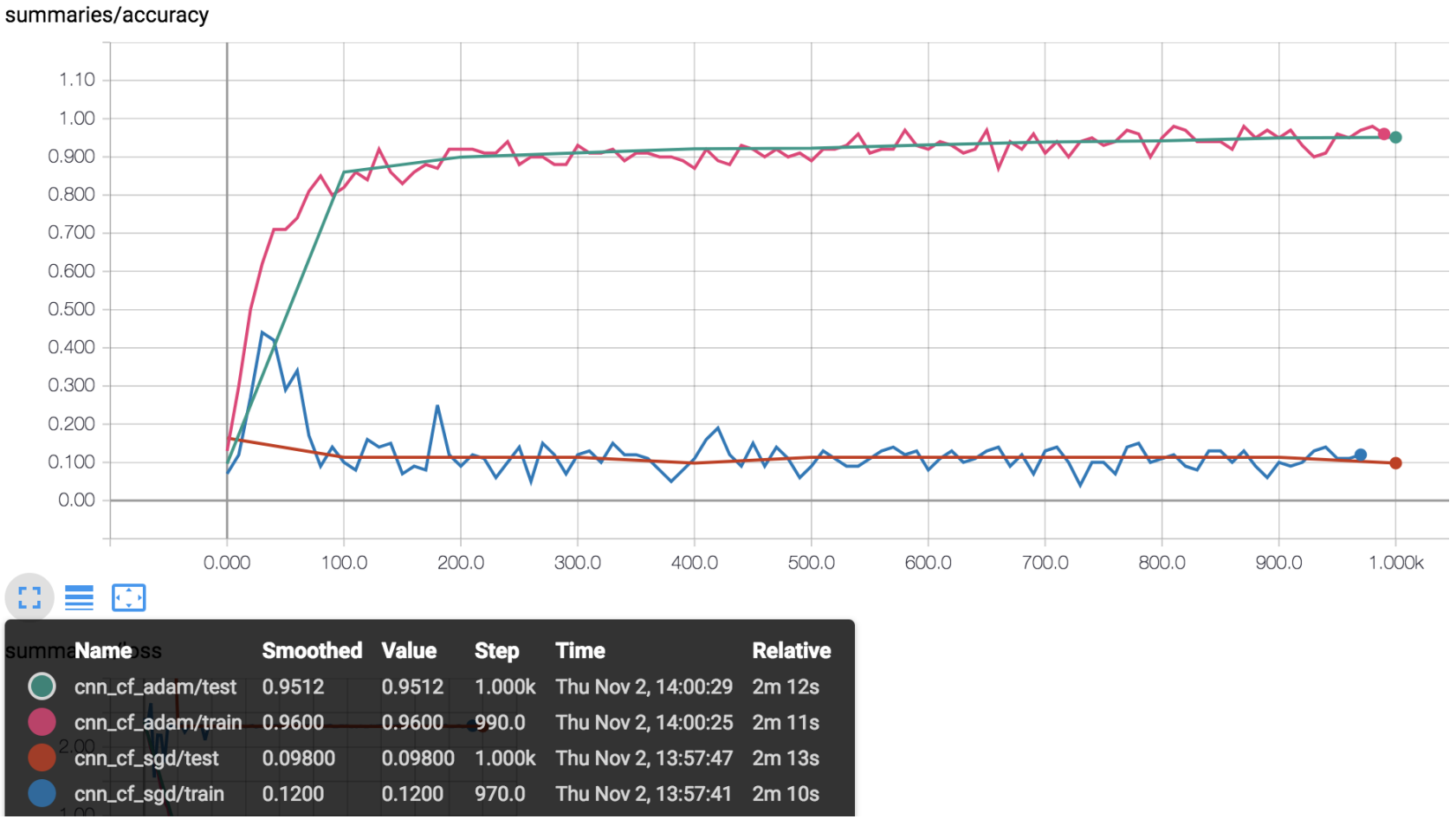
    ...or equivalently `tf.losses.softmax_cross_entropy`
  - `tf.nn.sparse_softmax_cross_entropy_with_logits`

    ...or equivalently `tf.losses.sparse_softmax_cross_entropy`
  - The former is for one hot encodings; the latter for $\{1, \ldots, K\}$ encodings of labels
  - Never write out the actual cross entropy equation

Fix it. Run it...

CAUTION: CHOICE OF OPTIMIZER

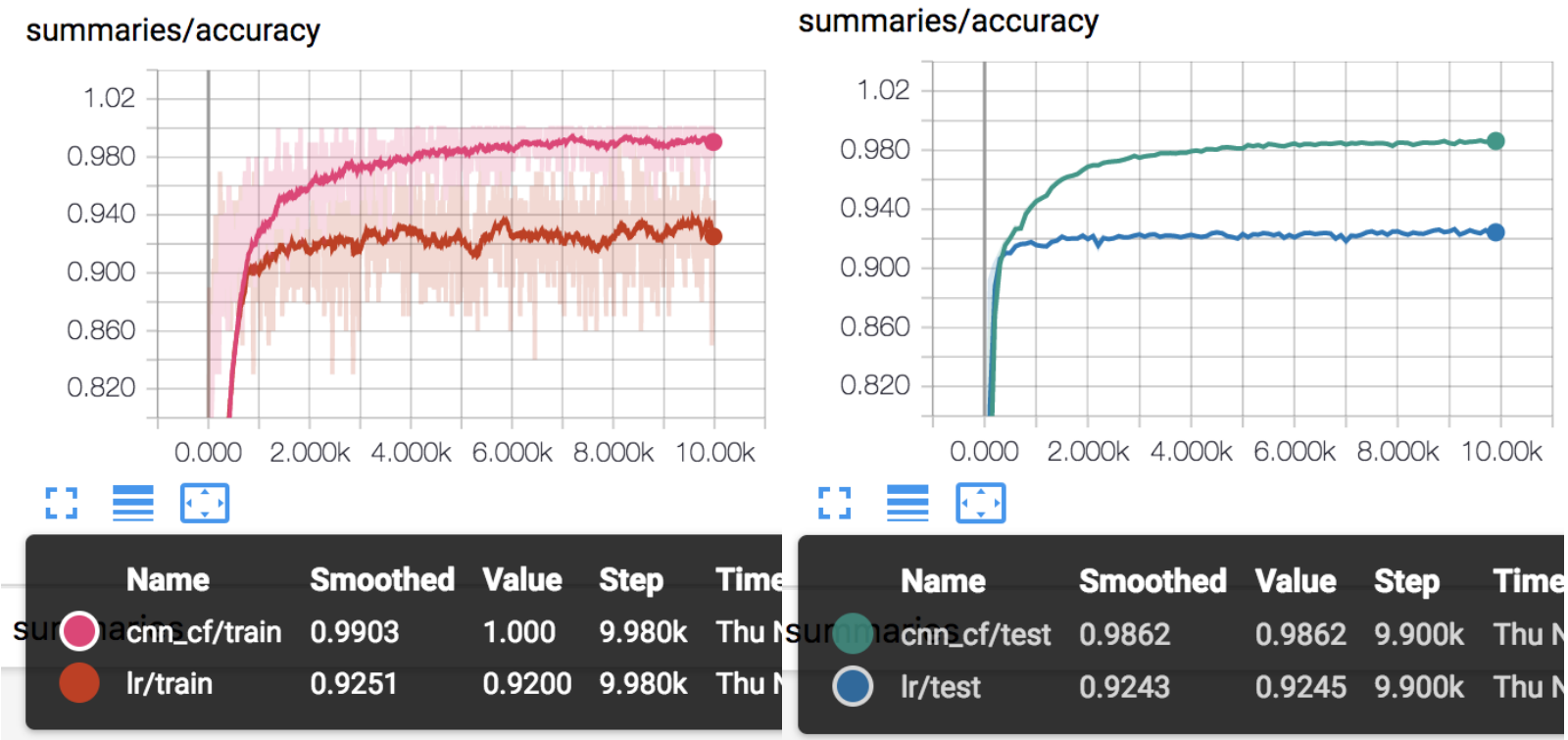
Consider different SGD variants



We will stick mostly with Adam for remainder, but again, empiricism...

PROGRESS WITH cnn_cf

Training and Test



Questions

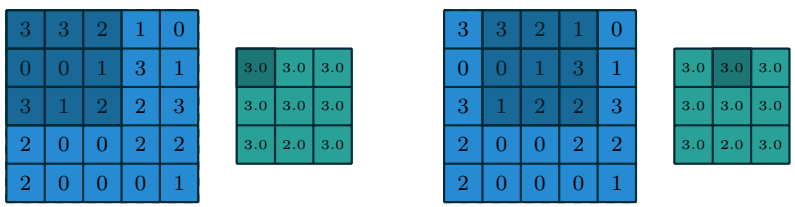
- Why is test/train nonsmooth/smooth?
- How do I set up tensorboard summaries for train and test?
- Will we do better if we make this network more complicated/deeper?
- Am I concerned by a $\approx 0.4\%$ difference between train and test?

TRICKS OF THE TRADE: POOLING

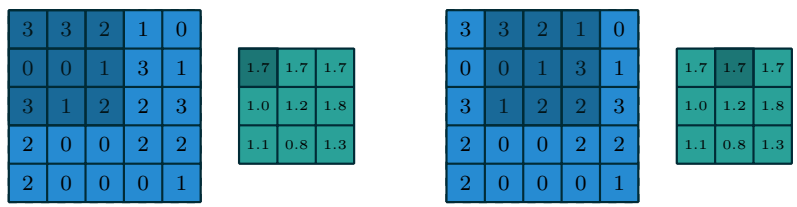
Idea

- Perhaps we care less about the precise location of activations in every layer
- And we know that parameters will be creeping upwards with padded layers
- *Pooling* adds a layer that averages or takes the max of a small window of activations
- Note: operates on each activation map individually
- Also called subsampling/downsampling (cf [Lecun et al 1998] figure earlier)

Max Pooling (most popular)



Average Pooling



Now

- I can reduce the number of parameters without (hopefully) losing much expressivity...
- I can increase the expressivity (hopefully) without increasing the number of parameters

ADDING COMPLEXITY

Make `cnn_cpcpff`: `conv→pool→conv→pool→fc→fc`

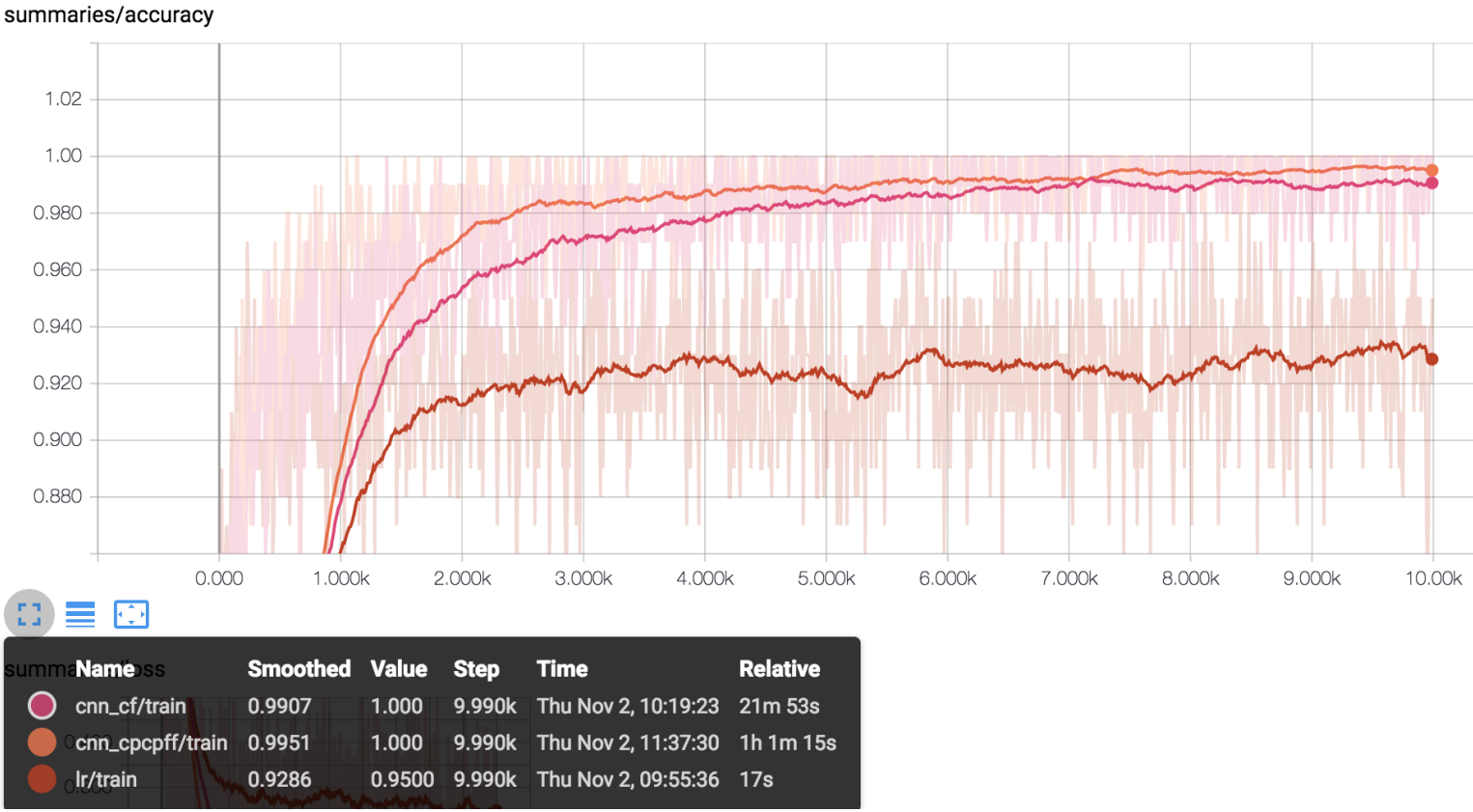
```
elif model_type=='cnn_cpcpff':
    # 2 layer cnn, similar architecture to tensorflow's deep mnist tutorial, so you can compare
    n1 = 32
    n2 = 64
    n3 = 1024
    x_image = tf.reshape(x, [-1,28,28,1]) # batch, then width, height, channels
    # cnn layer 1
    W_conv1 = tf.get_variable('W_conv1', shape=[5, 5, 1, n1])
    b_conv1 = tf.get_variable('b_conv1', shape=[n1])
    h_conv1 = tf.nn.relu(tf.add(conv(x_image, W_conv1), b_conv1))
    # pool 1
    h_pool1 = maxpool(h_conv1)
    # cnn layer 2
    W_conv2 = tf.get_variable('W_conv2', shape=[5, 5, n1, n2])
    b_conv2 = tf.get_variable('b_conv2', shape=[n2])
    h_conv2 = tf.nn.relu(tf.add(conv(h_pool1, W_conv2), b_conv2))
    # pool 2
    h_pool2 = maxpool(h_conv2)
    # fc layer to logits (7x7 since 2 rounds of maxpool)
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*n2])
    W_fc1 = tf.get_variable('W_fc1', shape=[7*7*n2, n3])
    b_fc1 = tf.get_variable('b_fc1', shape=[n3])
    h_fc1 = tf.nn.relu(tf.add(tf.matmul(h_pool2_flat, W_fc1), b_fc1))
    # one more fc layer
    # ... again, this is the logistic layer with softmax readout
    W_fc2 = tf.get_variable('W_fc2', shape=[n3,10])
    b_fc2 = tf.get_variable('b_fc2', shape=[10])
    logits = tf.add(tf.matmul(h_fc1, W_fc2), b_fc2, name='logits_cnn2')
```

Note:

- Draw this architecture
- Run it...

ADDING COMPLEXITY

Training performance



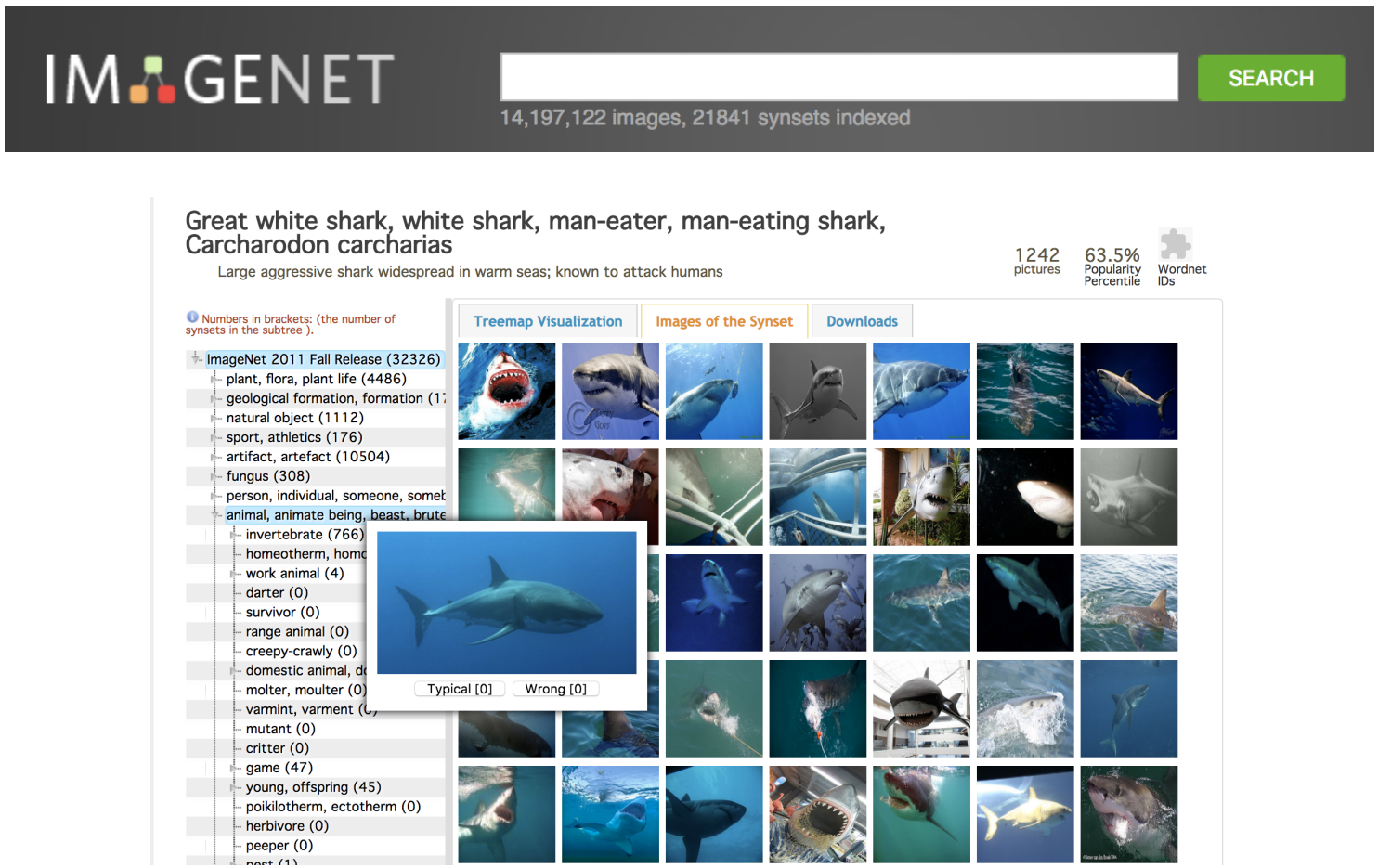
Name	Smoothed	Value	Step	Time	Relative
cnn_cf/train	0.9907	1.000	9.990k	Thu Nov 2, 10:19:23	21m 53s
cnn_cpcpff/train	0.9951	1.000	9.990k	Thu Nov 2, 11:37:30	1h 1m 15s
lr/train	0.9286	0.9500	9.990k	Thu Nov 2, 09:55:36	17s

Worth it?

- Better, but not much better.
- More costly

This story will change with more complex datasets...

The best large-scale vision dataset available

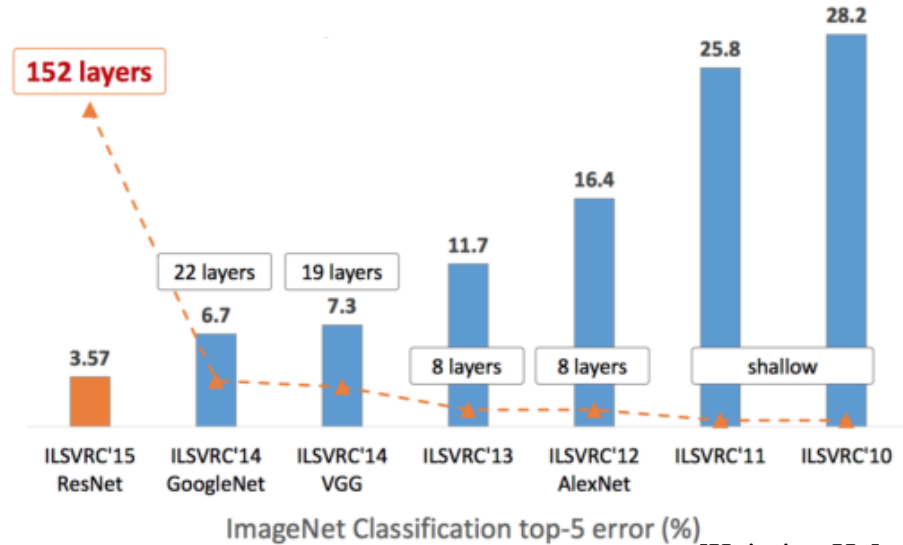


Note also that, in many images, bounding boxes are now provided

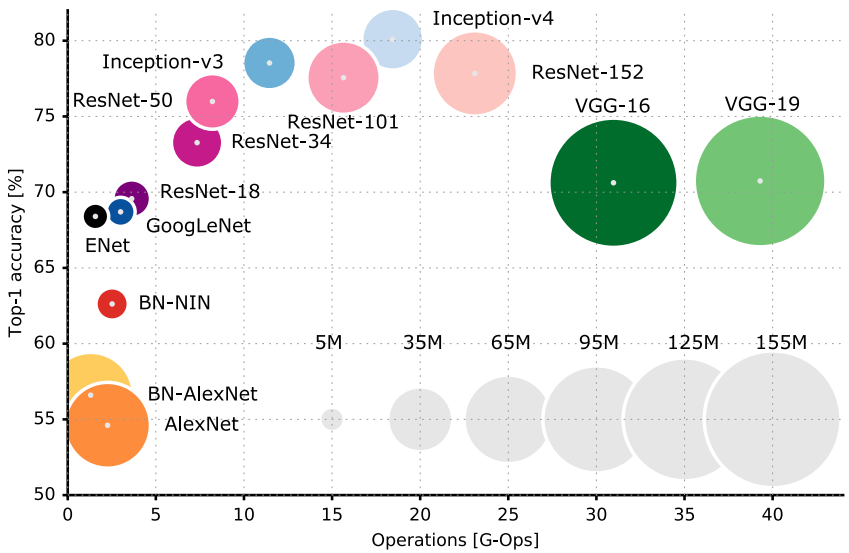
IMAGENET CHALLENGE

ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- Annual computer vision challenge
- e.g. ILSVRC 2014 had $>$ 1MM training, 50K validation, 100K test
- Multinomial classification $K = 1000$
- Since 2012, dominated by CNNs of increasing complexity
- Human performance surpassed in 2015
- Not without controversy...

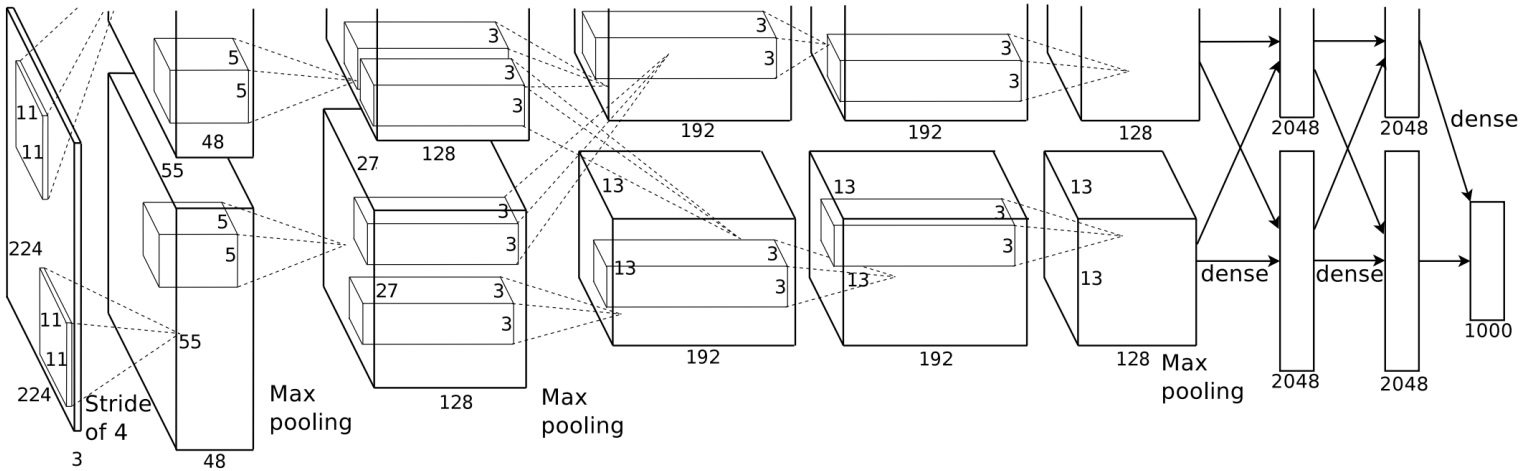


[Kaiming He]



[Canziani et al 2017]

The first ILSVRC winner with deep learning

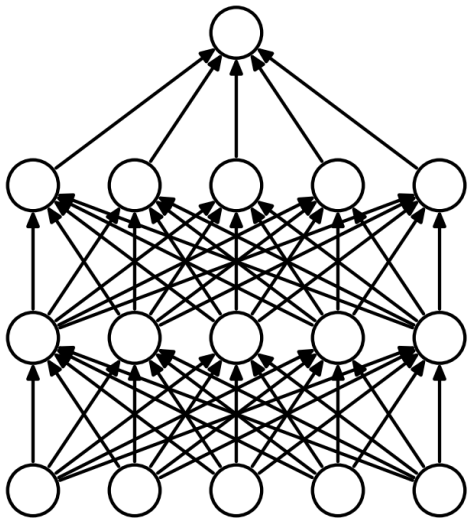


[Krizhevsky et al 2012]

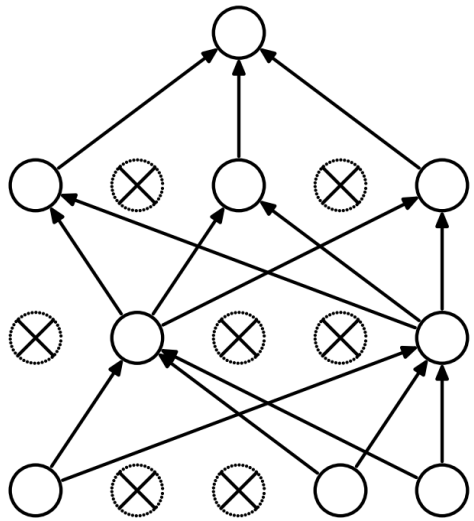
We can understand the entirety of this network

TRICKS OF THE TRADE: DROPOUT

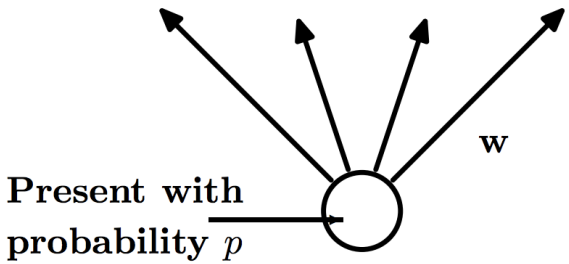
With increasing complexity comes increasing overfitting. Let's regularize!



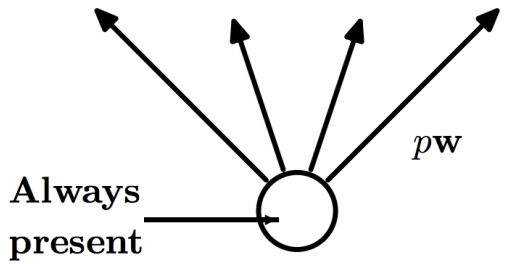
(a) Standard Neural Net



(b) After applying dropout.



(a) At training time



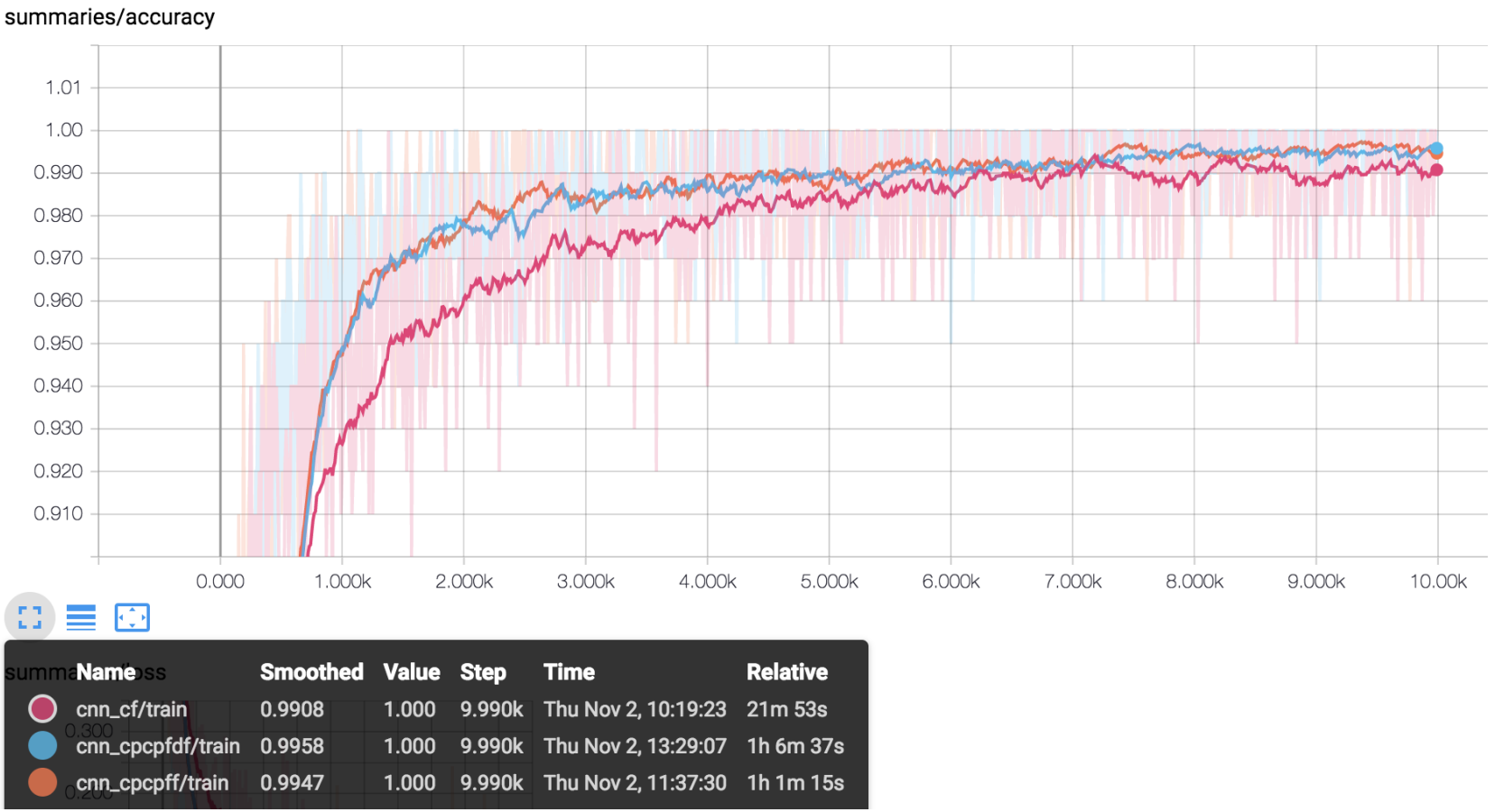
(b) At test time

[Srivastava et al 2014]

This widely used strategy is *dropout*

TRICKS OF THE TRADE: DROPOUT

Add a dropout layer: `conv→pool→conv→pool→fc→drop→fc`

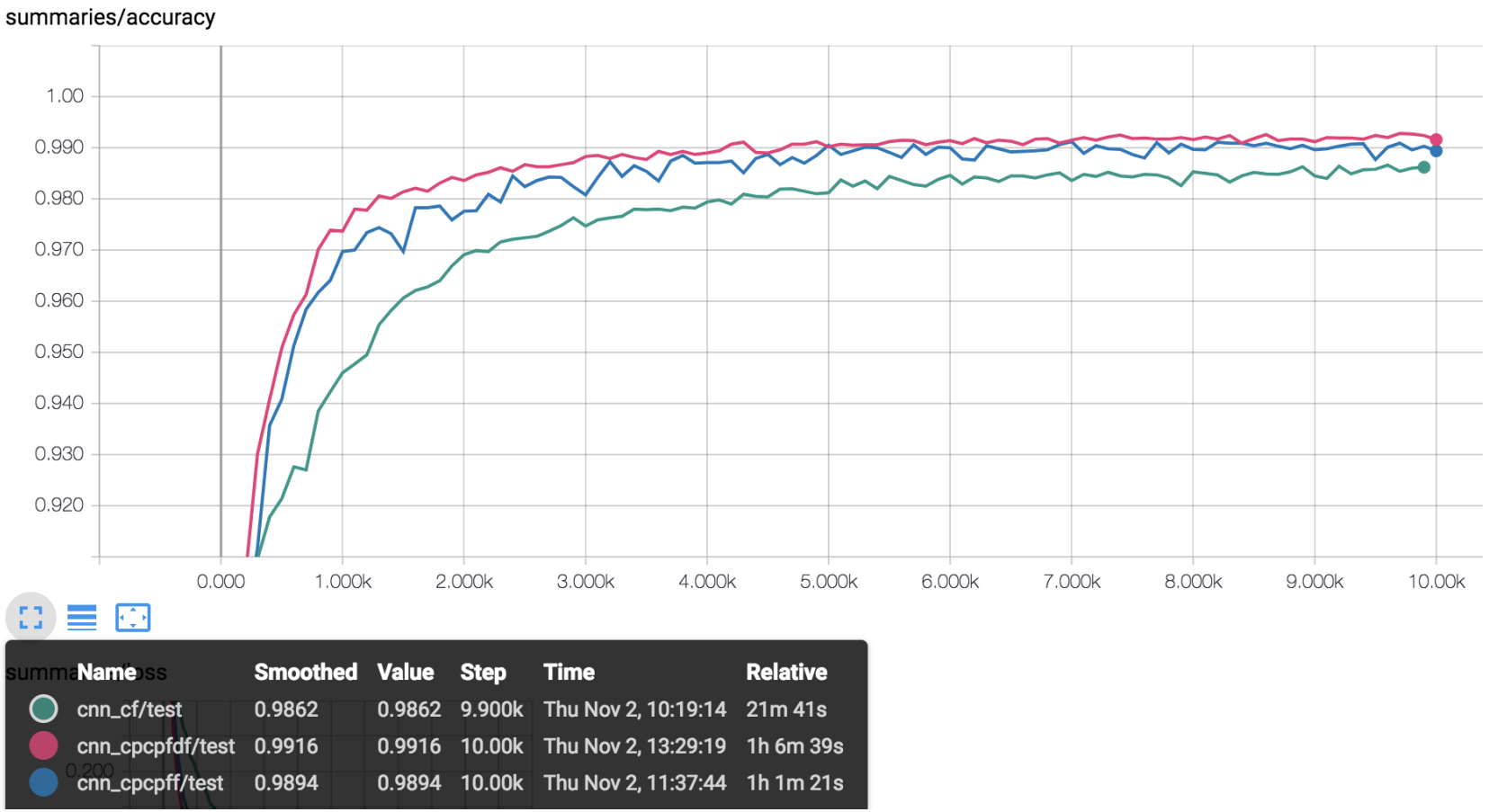


Name	Smoothed	Value	Step	Time	Relative
cnn_cf/train	0.9908	1.000	9.990k	Thu Nov 2, 10:19:23	21m 53s
cnn_cpcpfdf/train	0.9958	1.000	9.990k	Thu Nov 2, 13:29:07	1h 6m 37s
cnn_cpcpff/train	0.9947	1.000	9.990k	Thu Nov 2, 11:37:30	1h 1m 15s

Does not seem to affect training much...

TRICKS OF THE TRADE: DROPOUT

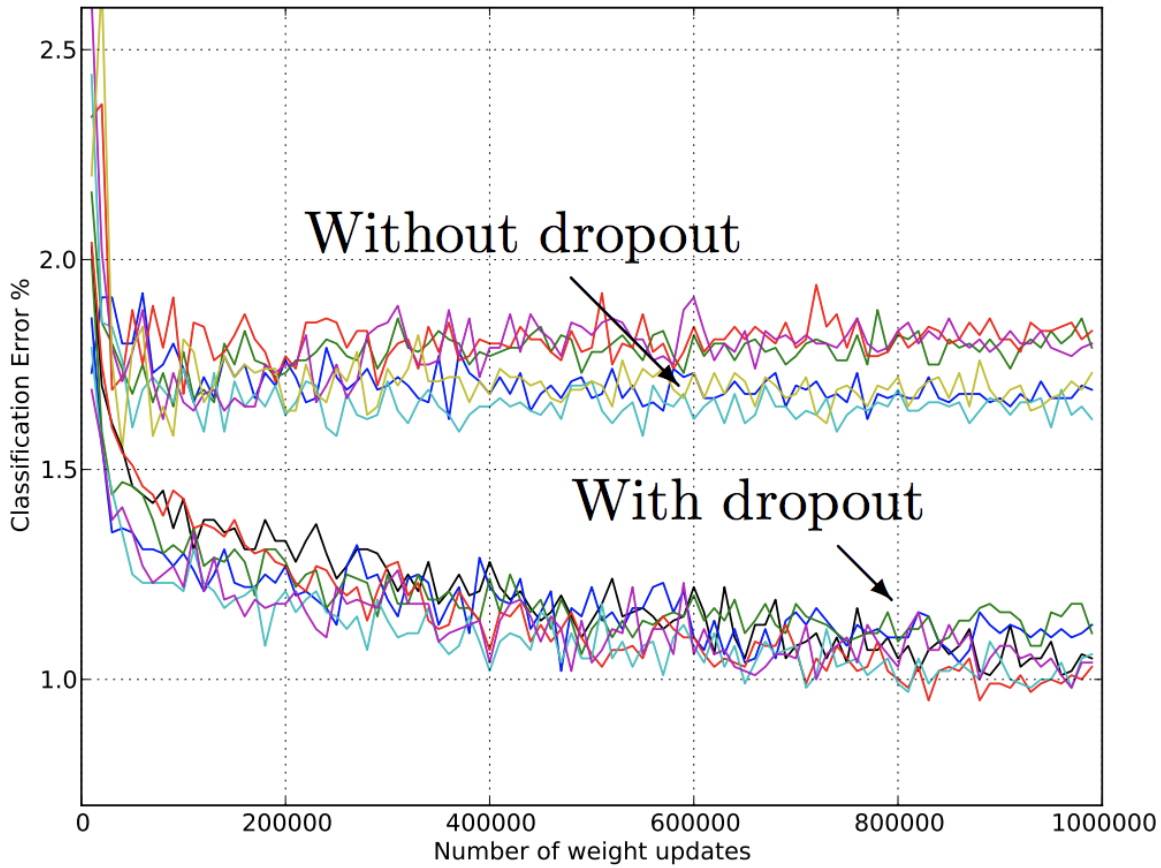
But hopefully it mitigates overfitting



Discuss... again, we expect this to matter more in more complex networks

TRICKS OF THE TRADE: DROPOUT

Dropout has become standard practice in modern network design



[Srivastava et al 2014]

STRONGLY RECOMMENDED!

Play with the architectures and choices we have made so far.
Experience is the only way to improve your deep learning skills.

Some ideas:

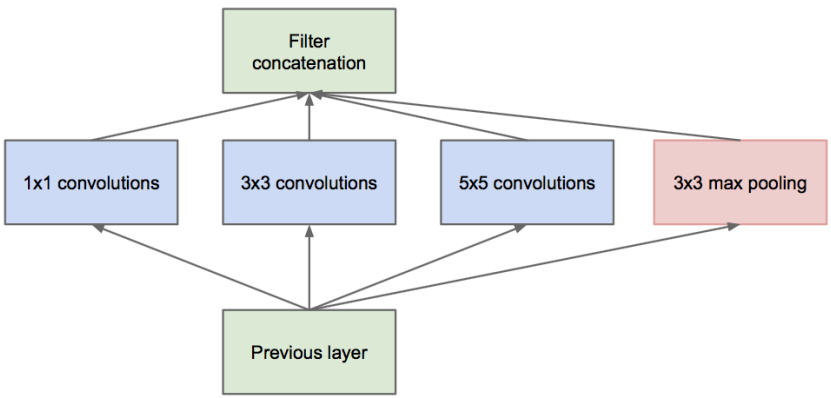
- Change the filters: sizes, striding, padding
- Change the pooling: average/max, different sizes, different positions
- Change the architecture
- Change the optimization method
- Change the batch size
- Change the summary/tensorboard content
- ...

INCEPTION MODULES

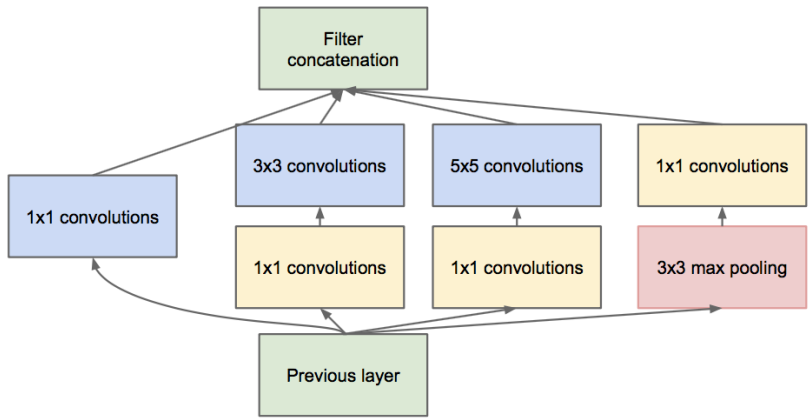
2014 ILSVRC winner added yet more complexity... Idea:

- Build a useful block or *module* of layers
- Layer those modules together

Inception module



(a) Inception module, naïve version



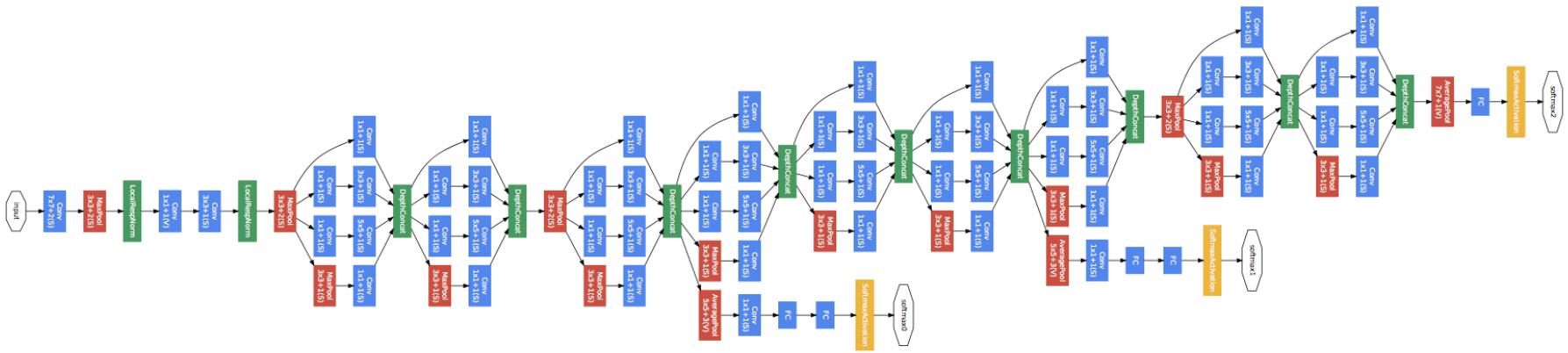
(b) Inception module with dimension reductions

[Szegedy et al 2014]

Reminder: 1×1 layers operate on the whole depth; act as dimension reduction

INCEPTION

Full network



[Szegedy et al 2014]

Notice auxiliary classifiers

- Concern: gradient info does not propagate deep into the network
- Not overfitting!
- A nice trick, but there is another that we will soon see

INCEPTION

Another view

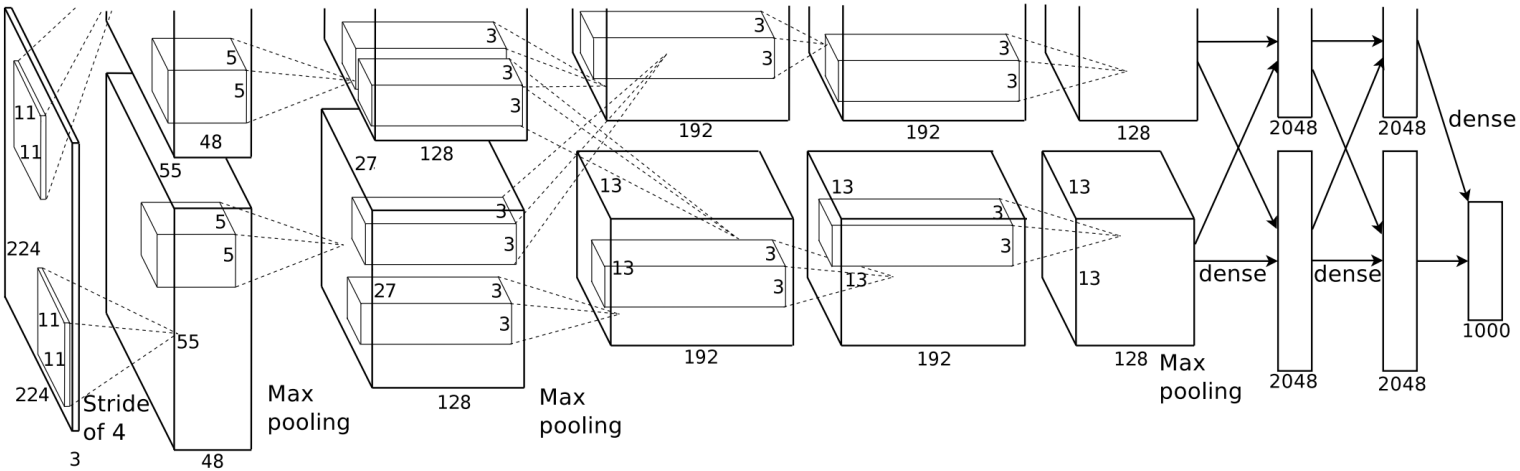
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

[Szegedy et al 2014]

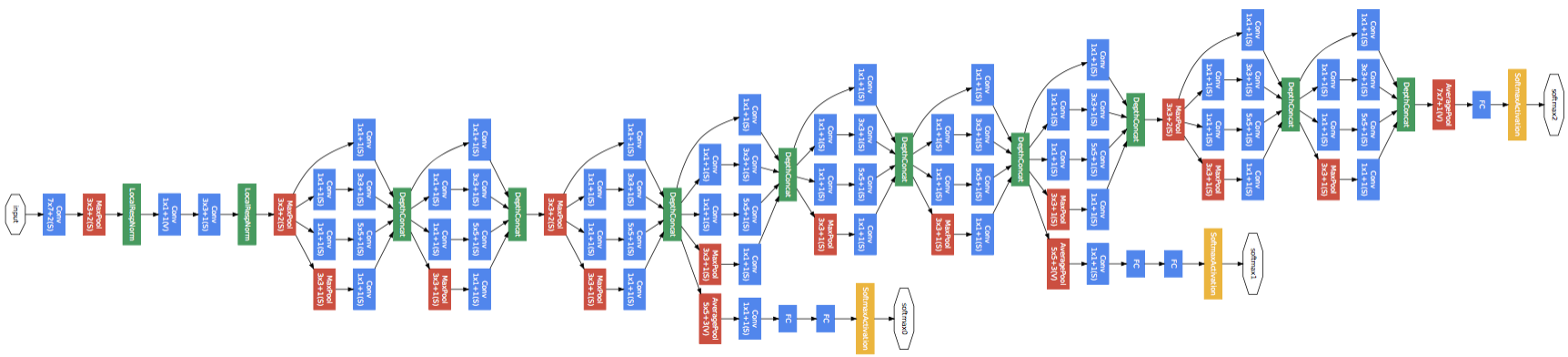
More complex, but still components we understand.

INTERLUDE: RETRAINING / TRANSFER LEARNING

Networks are trained for a specific task, but we suspect they also learn some useful concepts



[Krizhevsky et al 2012]

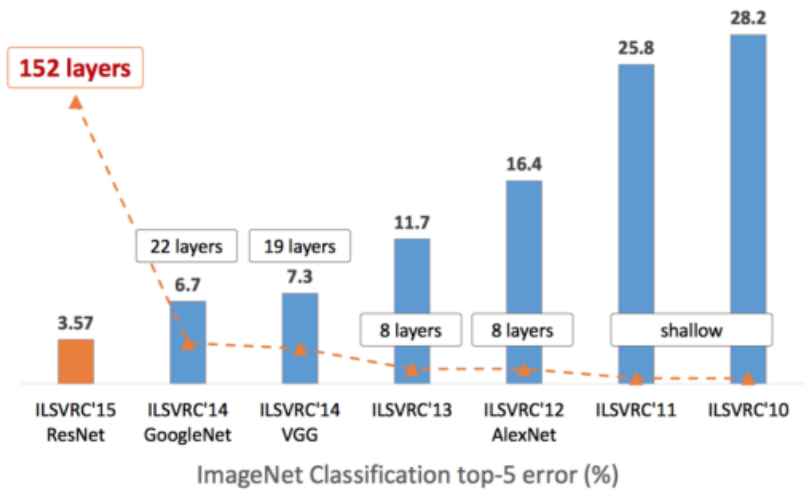


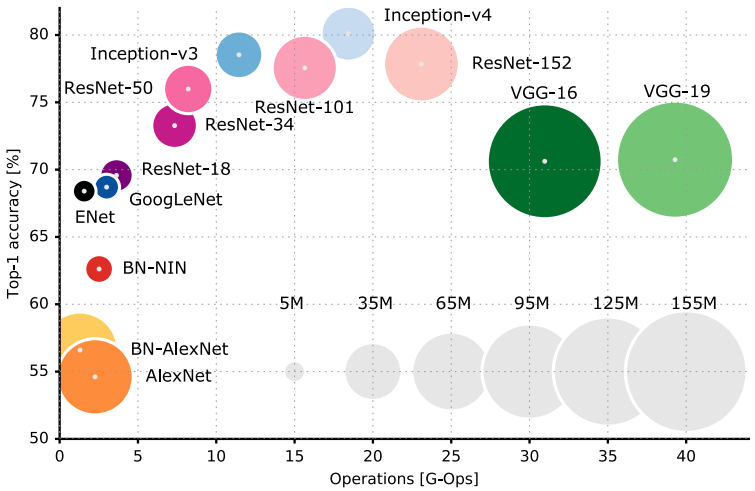
[Szegedy et al 2014]]

Idea: exploit a large pre-trained network to solve your problem...

2015 ILSVRC winner:

- added (vastly) more depth to the network
- successfully trained with one key idea
- surpassed human level performance
- did so with reasonably fewer parameters





[Kaiming He], [Canziani et al 2017]

PROBLEMS WTH DEPTH

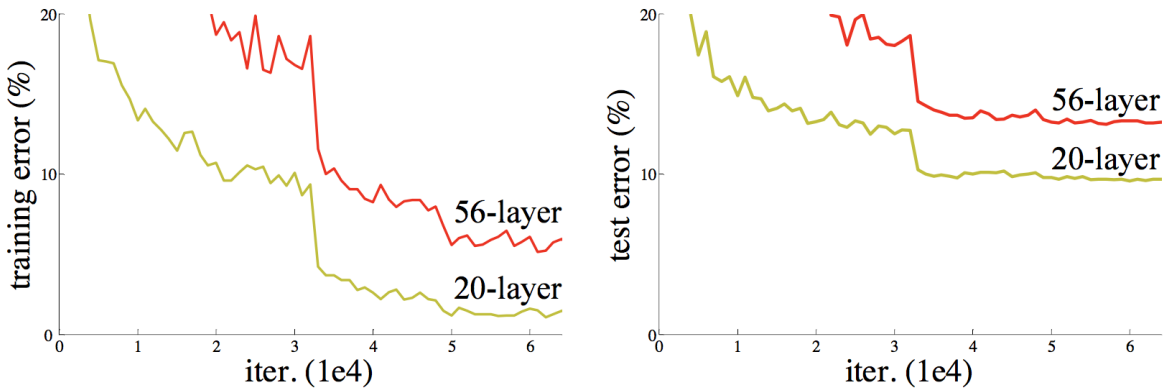
Exploding and *vanishing* gradients were a major historical problem for deep networks

- Chain rule has multiplicative terms, nonlinearities can saturate, etc.
- *Normalization* layers have been widely used to mitigate. Two popular strategies:
  - Local response norm.: divide unit activation by sum of squares of local neighbors

    [Krizhevsky et al 2012]
  - Batch norm.: standardize all units across the minibatch to a learned mean and var.

    [Ioffe and Szegedy 2015]
- Normalization is an important trick of the trade (as common as dropout and pooling)

Degradation has been another key roadblock to increasing depth



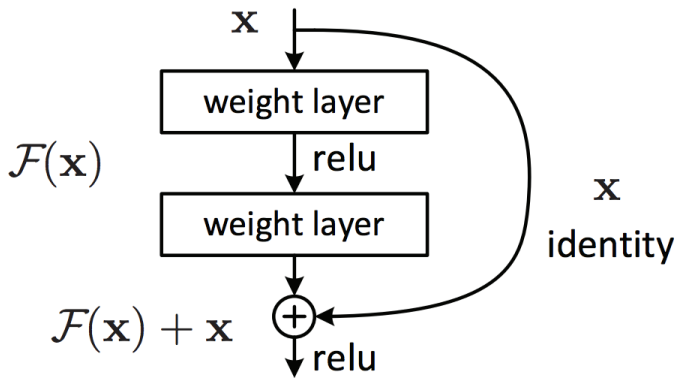
Notice:

- Training error *increasing* with *increasing* depth... not overfitting!
- Not an issue with the function family, since $\mathcal{F}_{20} \subset \mathcal{F}_{56}$
- Cause is optimization practicalities...

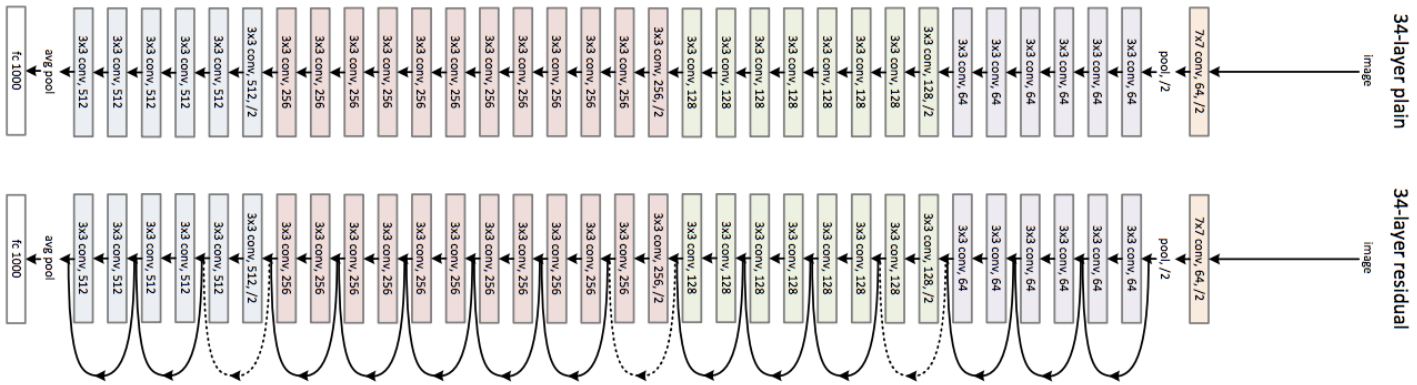
[He et al 2015]

ResNet

Key idea: layers learn residuals $x^{\ell+1} - x^{\ell}$ rather than the signal $x^{\ell+1}$ itself:



Layers naturally tend to identity transformation, degradation is avoided, large depth is enabled:



Resulting world leading performance, with many follow-on variations (layer dropout, e.g.)

[He et al 2015]

DEEP LEARNING REALITIES

MNIST $\rightarrow$ SVHN

Consider the same digit classification problem on (seemingly) similar data

MNIST vs. SVHN

Questions:

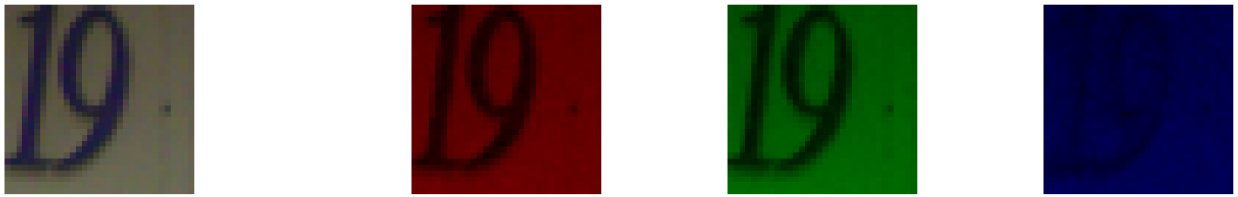
- If $\mathcal{F}$ was well chosen on MNIST, will it work well on SVHN?
- If yes, what does that mean?
- If no, what do we have to change to make it work?
- ...

- Key takeaway today: answering these questions is critical, hard, and very empirical
- We will go through a number of steps/lessons

1. DATA

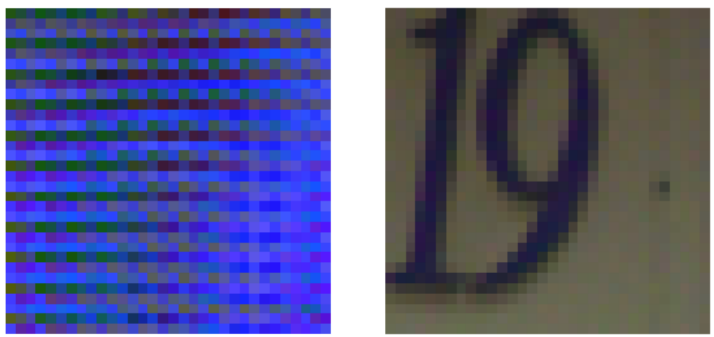
Input layer: three maps of size 32×32

$[32 \times 32 \times 3] = [32 \times 32 \times 1] \; [32 \times 32 \times 1] \; [32 \times 32 \times 1]$



- Check data to make sure it follows the labeling format you want (hint: it doesn't)
- Careful about reshaping in CNNs
- `tf` takes data from the first index of the input; is that an image?

```
4  x_re = X_train[:,:,:,batch].reshape([np.shape(batch)[0],-1])
5  # tf will then take this data one at a time from the first index.
6  xim = x_re[0,:]
7  # let us reshape and plot that to make sure it is correct
8  plot_save(xim.reshape([32,32,3]), 'svhn_c3')
9  # ugh that is not right...
10 # so we need to thoughtfully permute the indices of the tensor.  Get used to this and be careful.
11 plot_save(X_train[:,:,:,batch].transpose([3,0,1,2]).reshape([np.shape(batch)[0],-1])[0,:].reshape([32,32,3]), '
12 # better...
```



Run a simple model to get started...

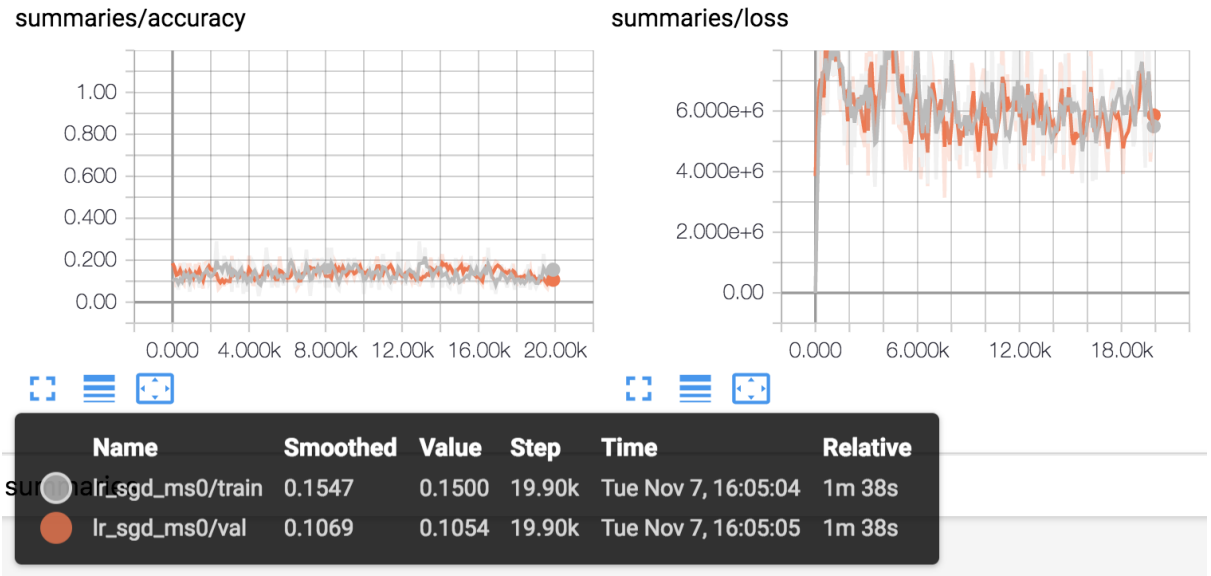
```
-> 1317                              options, run_metadata)
   1318     else:
   1319       return self._do_call(_prun_fn, self._session, handle, feeds, fetches)

~/anaconda/envs/aml_sandbox/lib/python3.6/site-packages/tensorflow/python/client/session.py in _do_
gs)
   1334         except KeyError:
   1335           pass
-> 1336       raise type(e)(node_def, op, message)
   1337
   1338   def _extend_graph(self):

InvalidArgumentError: Nan in summary histogram for: summaries/logits
         [[Node: summaries/logits = HistogramSummary[T=DT_FLOAT, _device="/job:localhost/replica:0/
0"](summaries/logits/tag, model/logits_cnn_cf)]]

Caused by op 'summaries/logits', defined at:
  File "/Users/jpc/anaconda/envs/aml_sandbox/lib/python3.6/runpy.py", line 193, in _run_module_as_m
    "__main__", mod_spec)
  File "/Users/jpc/anaconda/envs/aml_sandbox/lib/python3.6/runpy.py", line 85, in _run_code
```

Reminder!

- Be careful of numerical underflow and overflow; things like log 0 will crash your code with `NaN` errors (possibly just in `tb`)
- Numerical stability is always a concern in practical machine learning
- Again, always use:
  - `tf.nn.softmax_cross_entropy_with_logits`
  - similar numerically safe functions when in a related situation.

Fix it. Run it...

Start with logistic regression and SGD

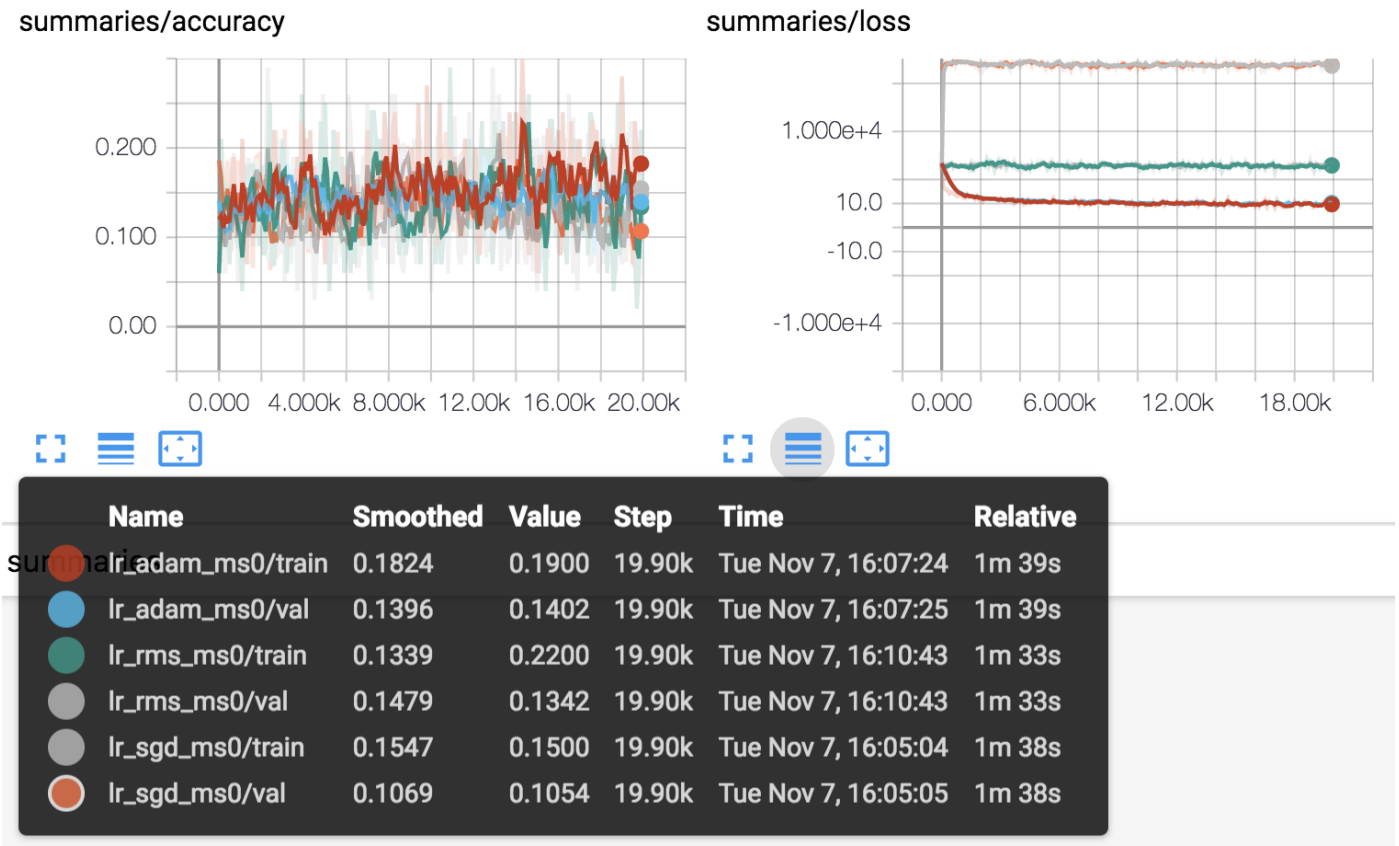


`tb` helps, but basic debugging is still useful

```
Step 200: training accuracy 0.1270
    sample pred: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
    sample true: [1 9 2 3 2 5 9 3 3 1 3 3 2 8 7 4 4 1 2 8]
    correct predictions by class: [  0   0 125   0   0   0   0   0   0   2]
Step 200: val accuracy 0.1328
Step 300: training accuracy 0.0600
    sample pred: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
    sample true: [1 9 2 3 2 5 9 3 3 1 3 3 2 8 7 4 4 1 2 8]
    correct predictions by class: [60  0  0  0  0  0  0  0  0  0]
Step 300: val accuracy 0.0652
Step 400: training accuracy 0.2060
    sample pred: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
    sample true: [1 9 2 3 2 5 9 3 3 1 3 3 2 8 7 4 4 1 2 8]
    correct predictions by class: [  0 201   0   0   0   5   0   0   0   0]
Step 400: val accuracy 0.1876
```

Not learning...

Switching from SGD to Adam has helped before; we'll also try RMSProp



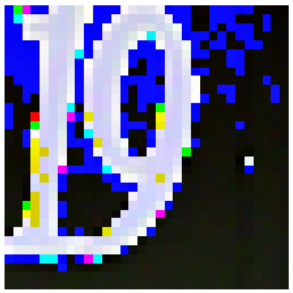
Name	Smoothed	Value	Step	Time	Relative
lr_adam_ms0/train	0.1824	0.1900	19.90k	Tue Nov 7, 16:07:24	1m 39s
lr_adam_ms0/val	0.1396	0.1402	19.90k	Tue Nov 7, 16:07:25	1m 39s
lr_rms_ms0/train	0.1339	0.2200	19.90k	Tue Nov 7, 16:10:43	1m 33s
lr_rms_ms0/val	0.1479	0.1342	19.90k	Tue Nov 7, 16:10:43	1m 33s
lr_sgd_ms0/train	0.1547	0.1500	19.90k	Tue Nov 7, 16:05:04	1m 38s
lr_sgd_ms0/val	0.1069	0.1054	19.90k	Tue Nov 7, 16:05:05	1m 38s

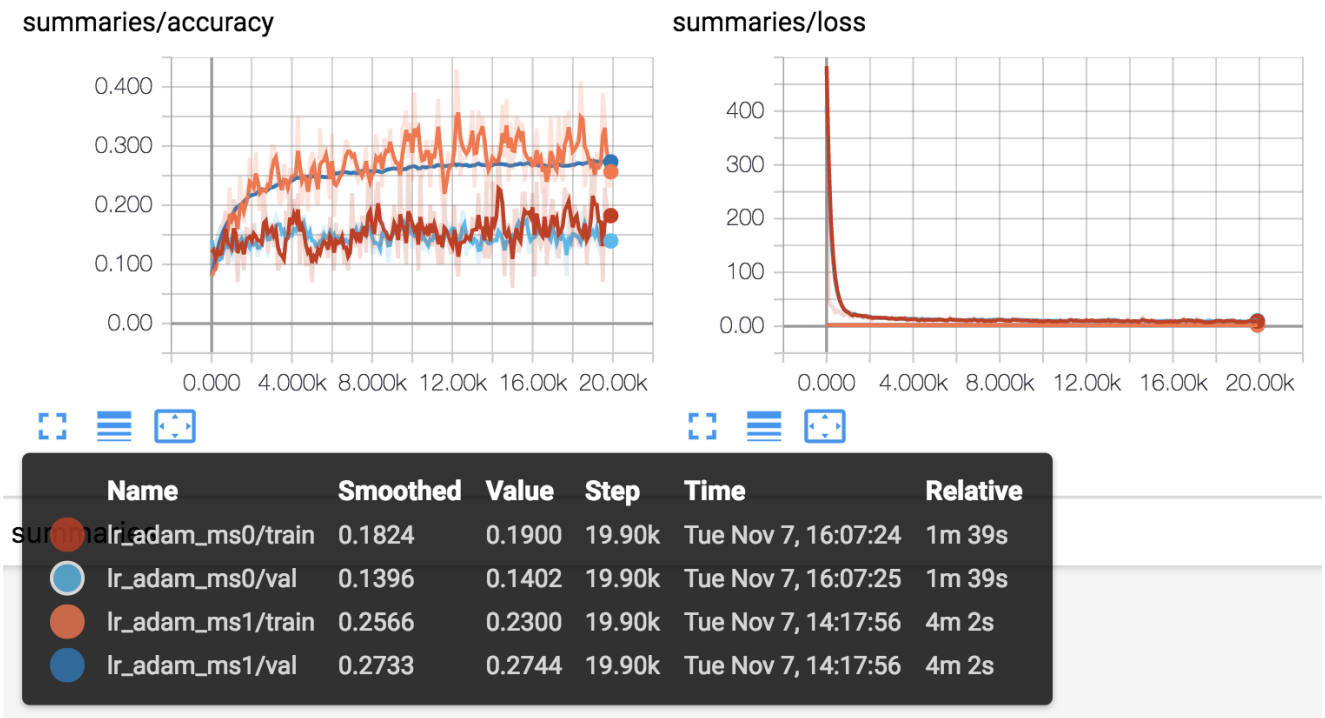
Performance is still terrible, but at least the loss function is not pathological. Progress...

5. MEAN SUBTRACTION

Observation

- SVHN data has very different illumination/brightness
- Precondition via mean subtraction of each channel?

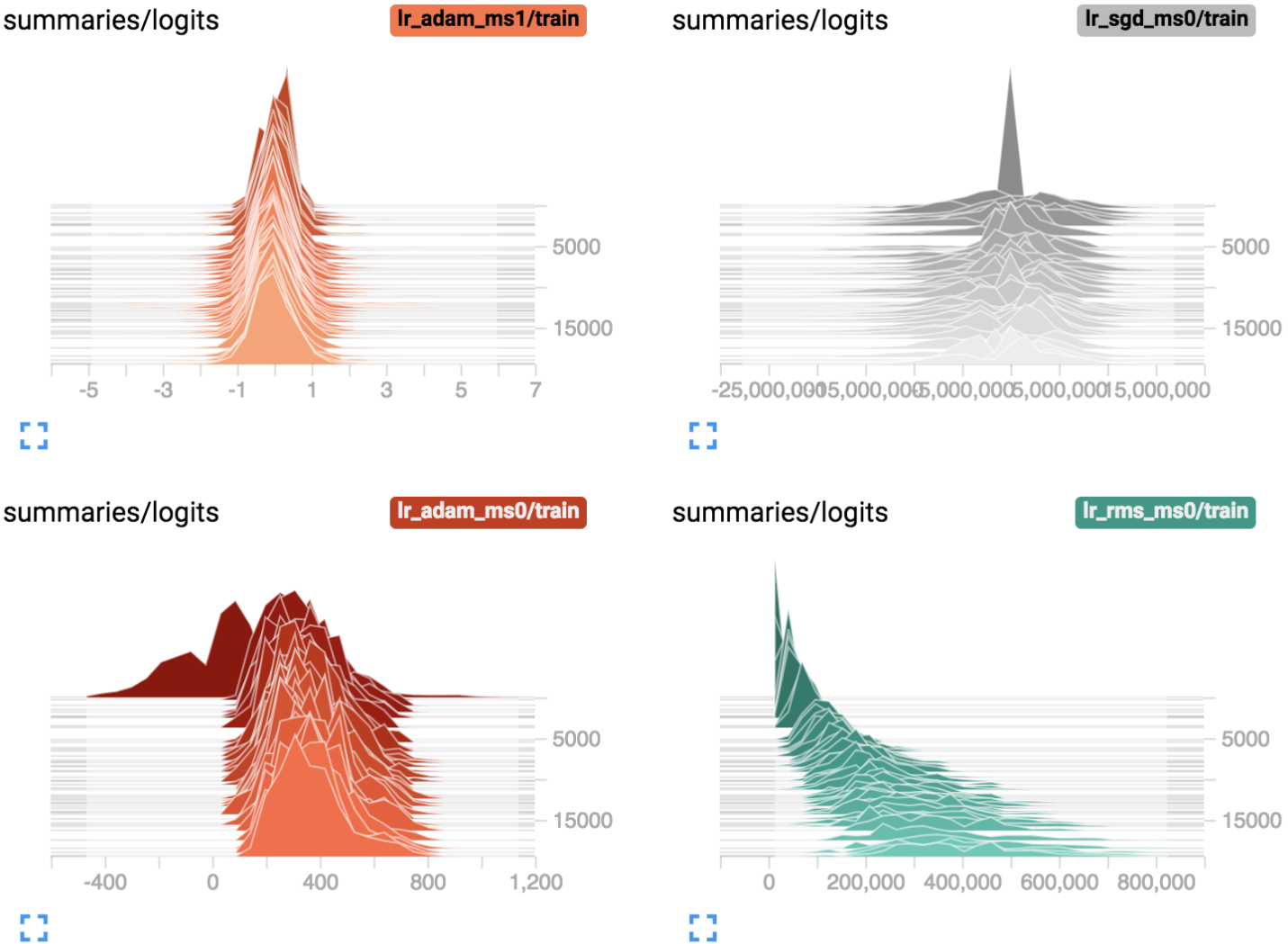




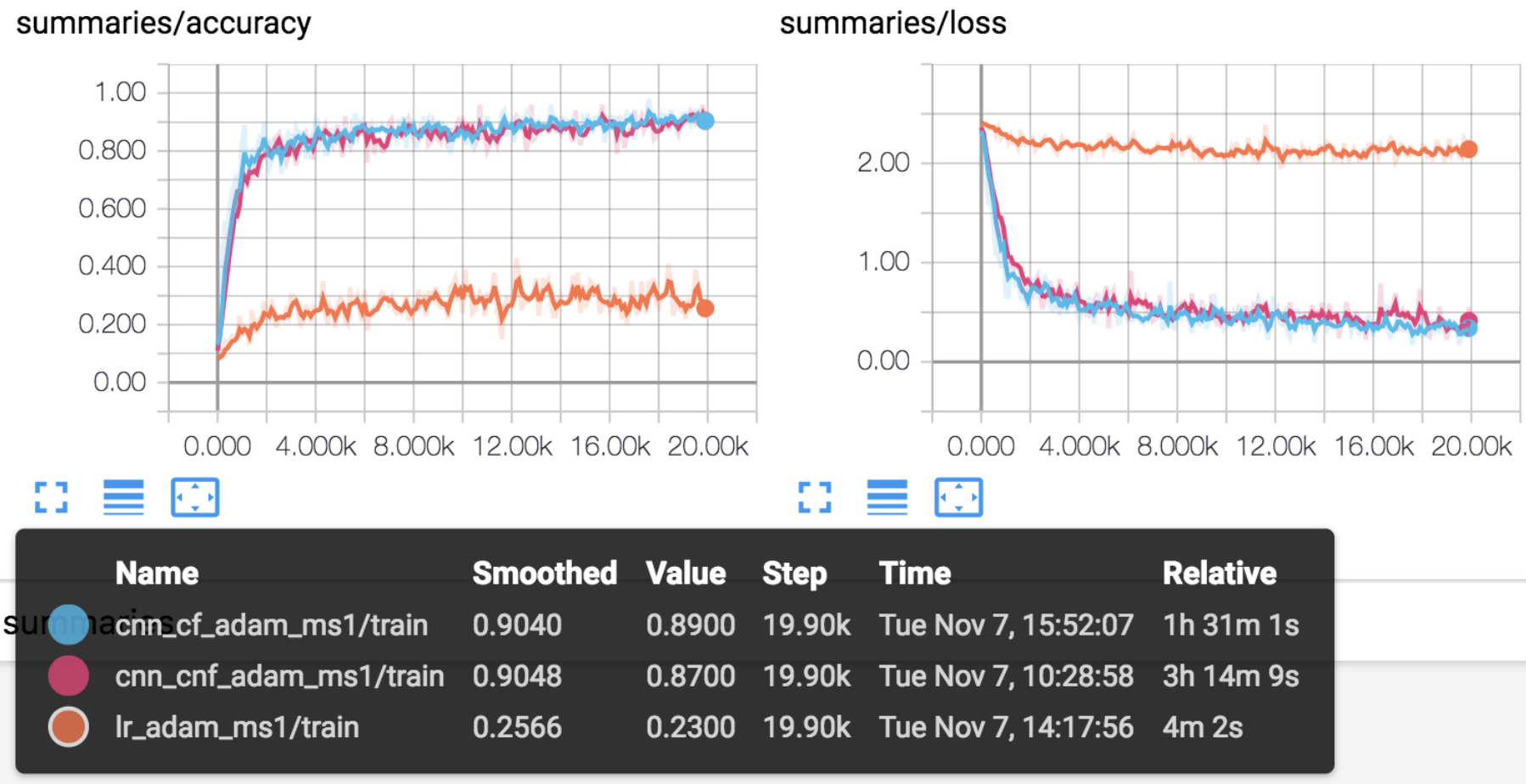
Progress! Preprocessing data matters... do not rely on the neural net to do all the work

6. TENSORBOARD FOR EMPIRICISM

Look at the histograms of logits over time to choose which one is learning.

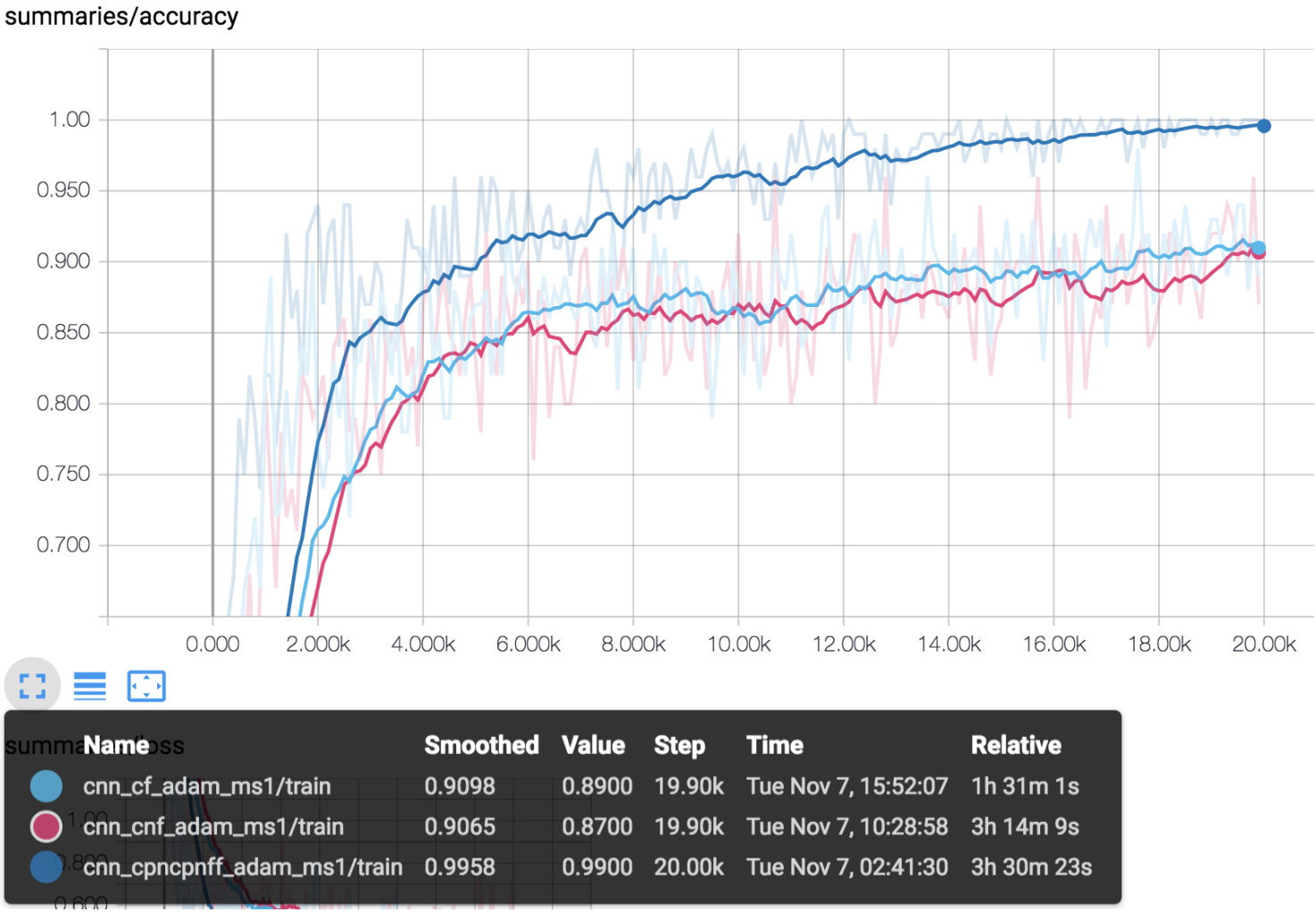


Add `cnn_cf`: `conv`→ `fc` and `cnn_cnf`: `conv`→ `norm` → `fc`



7. ADDING COMPLEXITY

Add `cnn_cpncpnff`: `conv`→`pool`→`norm`→ `conv`→`pool`→`norm`→`fc`→`fc`

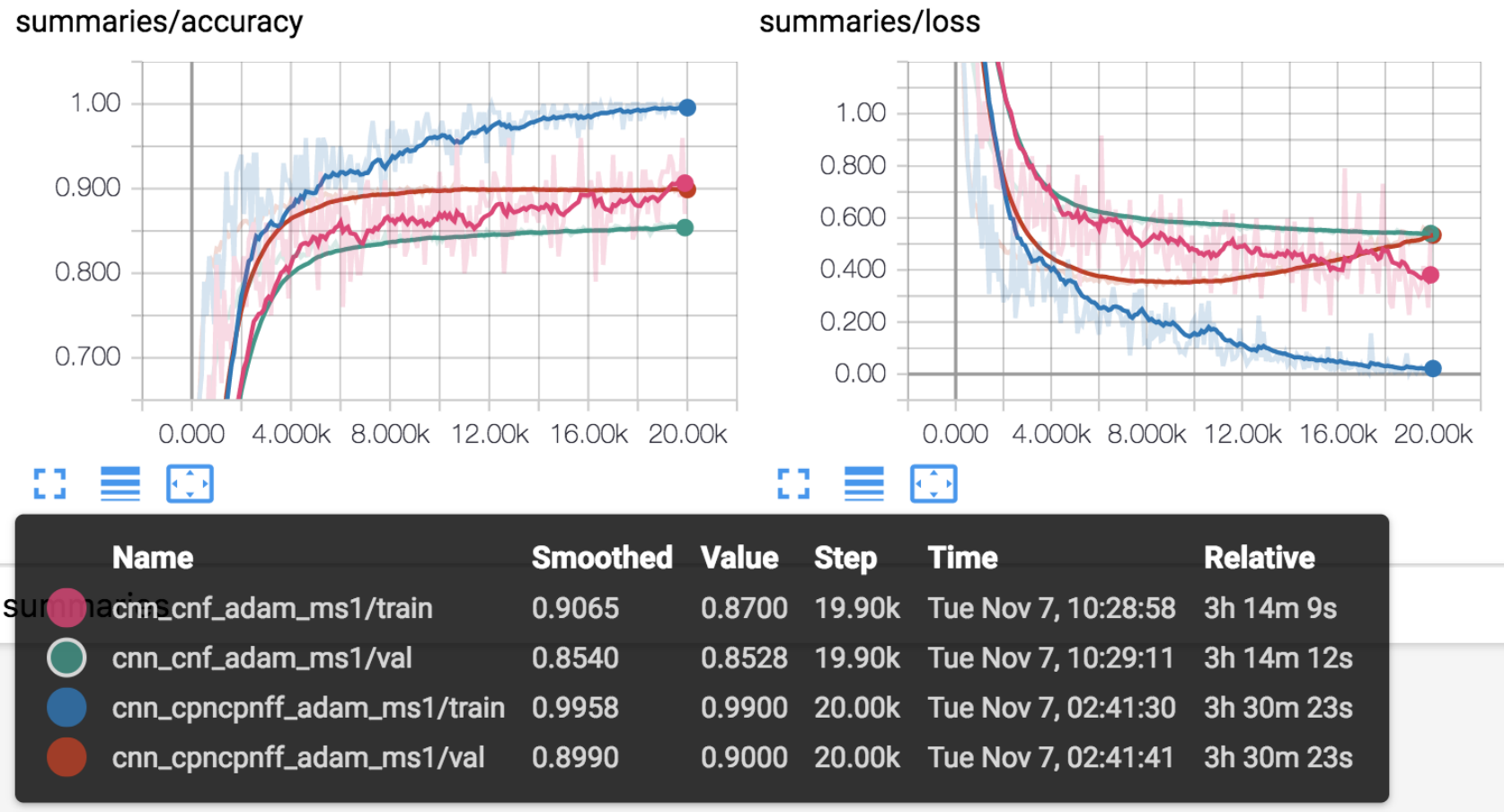


Training performance is very high. Overfitting?

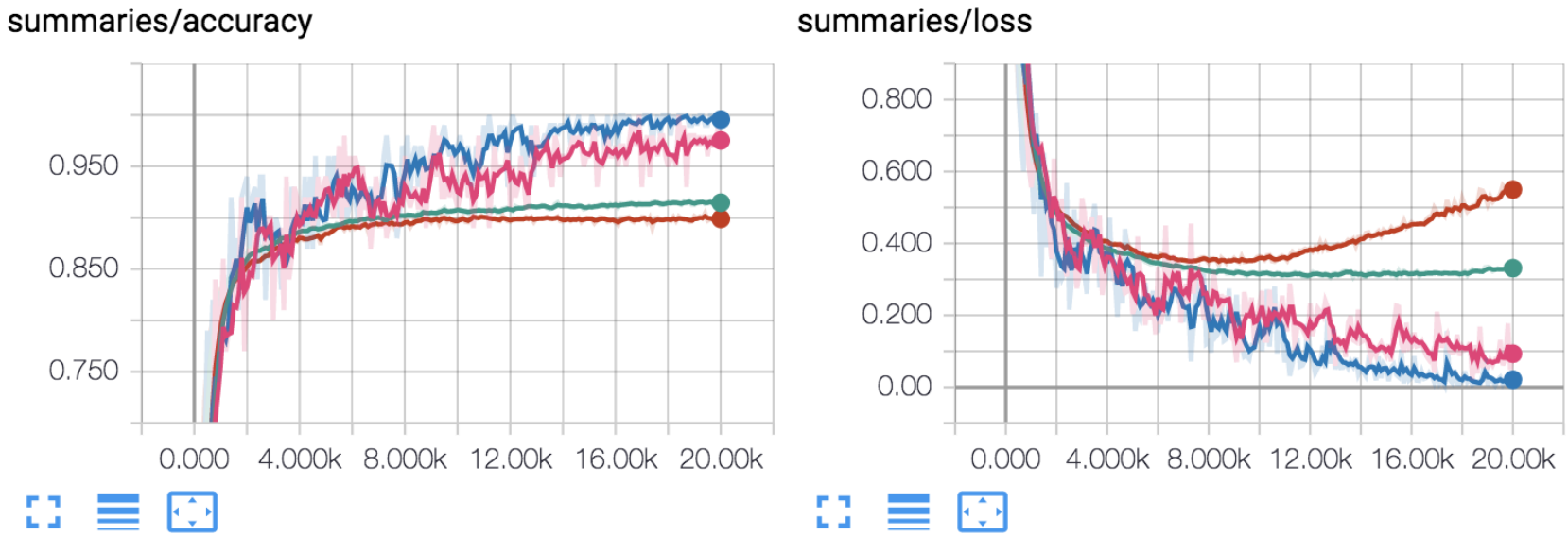
8. VALIDATION DATA

A separate validation set:

- helps monitor training
- avoids data snooping (overfitting to the test set)
- clarifies overfitting (substantial here!)



Add a dropout layer to regularize



summaries

Name	Smoothed	Value	Step	Time	Relative
cnn_cpncpnfdf_adam_ms1/train	0.9752	0.9700	20.00k	Tue Nov 7, 22:38:31	3h 30m 47s
cnn_cpncpnfdf_adam_ms1/val	0.9146	0.9148	20.00k	Tue Nov 7, 22:38:42	3h 30m 46s
cnn_cpncpnff_adam_ms1/train	0.9956	0.9900	20.00k	Tue Nov 7, 02:41:30	3h 30m 23s
cnn_cpncpnff_adam_ms1/val	0.8989	0.9000	20.00k	Tue Nov 7, 02:41:41	3h 30m 23s

10. HYPERPARAMETER SEARCH

To further improve performance, carefully search the free (hyper)parameters:

- Change the filters
- Change the architecture
- Change the optimization method
- Change the parameters of those methods (Adam learning rate, dropout prob, etc.)
- Scrutinize mislabels to look for patterns
- Be mindful of overfitting, including overfitting to your validation set
- ...

> Excellence in deep learning comes from experience and empiricism.

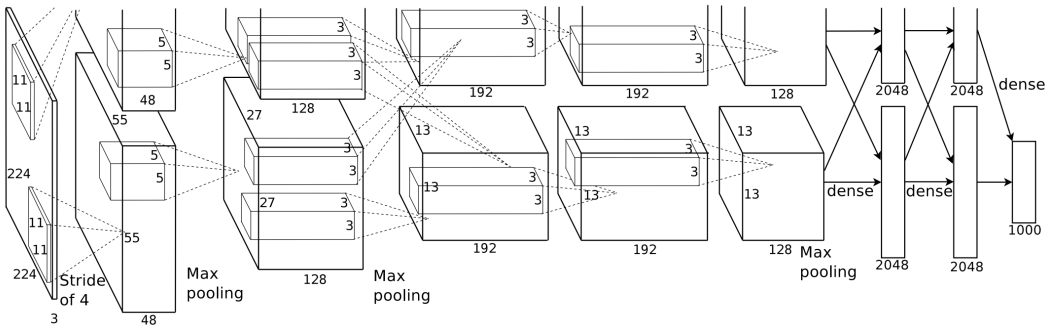
Tools and tricks at your disposal:

- Convolutional layers: filter size, zero padding, striding
- Optimization: SGD, Adam, RMSProp, etc.
- Intermediate layers: pooling, dropout, normalization
- Monitoring: validation data, tensorboard, classic debugging

Summarizing Convolutional Neural Networks

Convolutional neural networks are the power beind modern computer vision

- The idea of a convolution saves parameters and exploits knowledge of local statistics
- In challenging datasets, CNNs produce excellent results
- They require much care and attention to be performant
- Deeper networks can achieve superhuman classification performance
- A particular architecture can be (very) problem specific



Discuss: is this *general/full* AI or weak/narrow/applied AI?

- Have we solved digit recognition, or simply MNIST and SVHN (separately)?
- How much more general is the problem of full computer vision?
- What about object recognition, multi-object tracking, video, prediction, etc.?

Reinforcement Learning

TRANSITION TO RL

Supervised learning

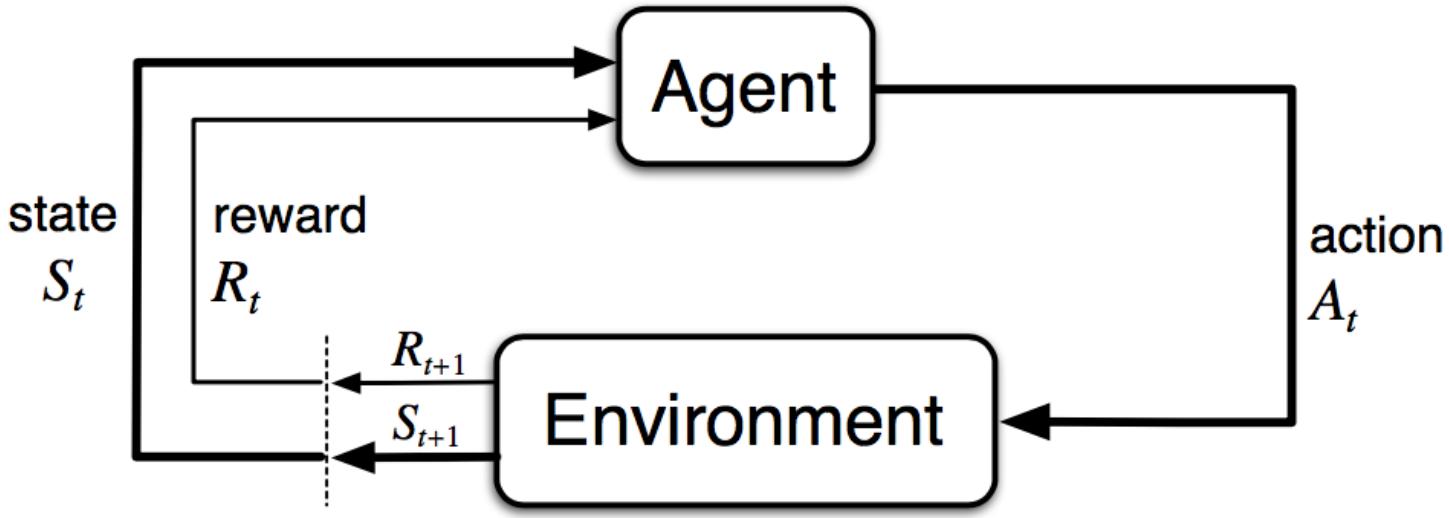
- is learning a relationship between iid input-output pairs
- relies on training data: examples of correct situations (e.g. an input image) along with the correct action (e.g. output the label ‘3‘)
- depends on this data being representative of all possible scenarios
- uses *instructive* feedback: it indicates the correct action regardless of what action is taken

Reinforcement learning

- is learning how to map an input situation into an output action to maximize reward
- relies on interaction: actions must explore possible actions, searching for good behavior
- operates where all possible scenarios can not reasonably be captured by training data
- uses *evaluative* feedback: training information evaluates value of actions taken

REINFORCEMENT LEARNING

Reinforcement Learning is the study of problems that can be characterized by...



...an *Agent*...

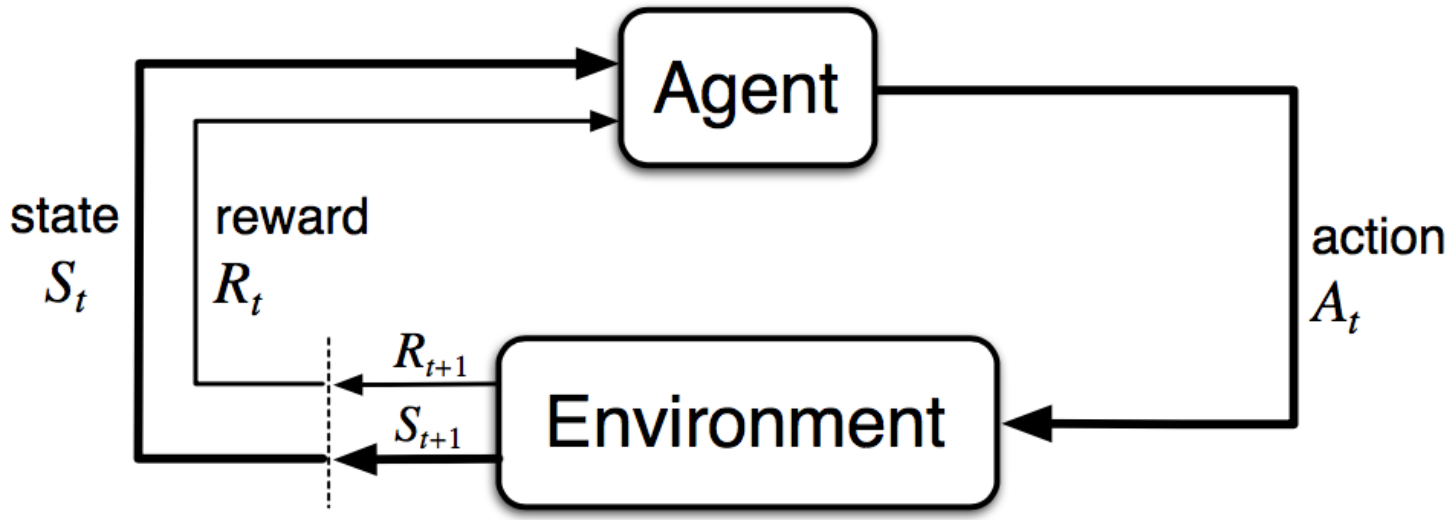
- takes *action* A_t at time t
- receives *reward* R_t
- observes *state* S_t

...interacting with an *environment*.

- affected by actions A_t
- produces rewards R_t
- updates its state S_t based on the agent's actions

REINFORCEMENT LEARNING

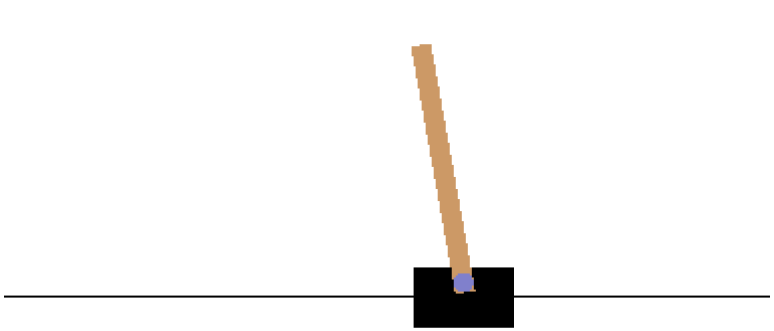
Reinforcement Learning is the study of problems that can be characterized by...

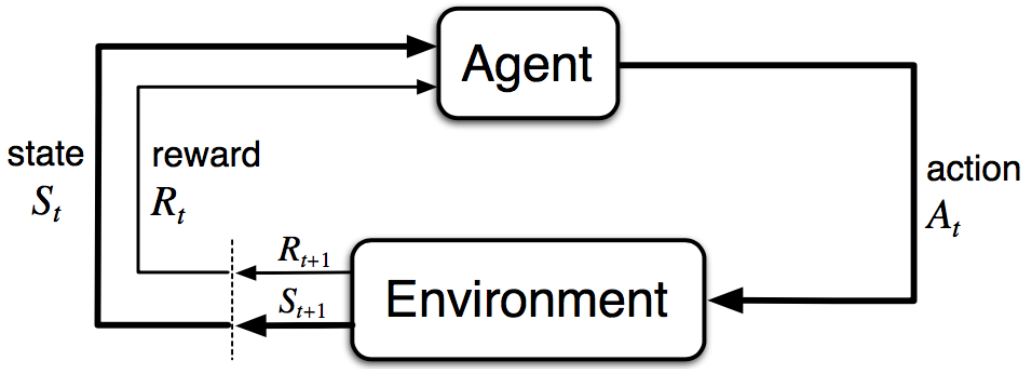


Note

- behavior will amount to a *policy* $\pi(a|s)$: the probability of taking action a when in state s
- state can be unchanging (this lecture), fully observed (next lectures), partially observed
- decision-making agent interacts with environment to achieve a goal (e.g. max reward)
- usually RL agents have to operate in presence of major uncertainty
- correct actions require planning and understanding future consequences of present action

EXAMPLE: CART POLE





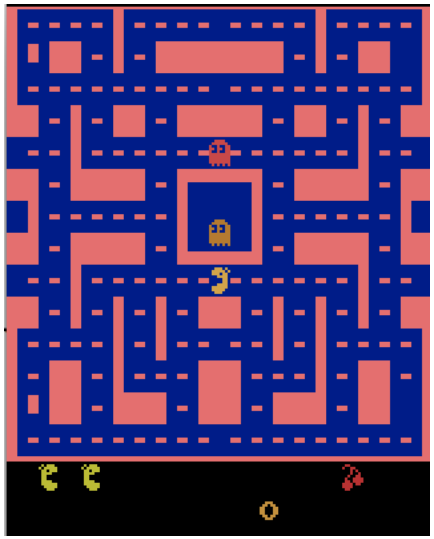
Example choices:

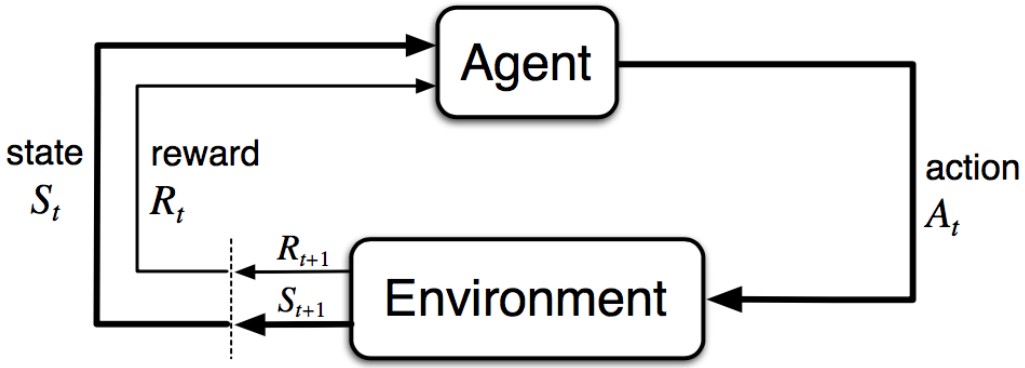
- State $S_t = [x_t, \dot{x}_t, \theta_t, \dot{\theta}_t]$, the position/velocity of cart; angle/velocity of pole
- Reward $R_t = +1$ if $\theta \in [\pi/2 - \alpha, \pi/2 + \alpha]$, $R_t = 0$ otherwise
- Action $A_t = \{\leftarrow, \rightarrow\}$; move cart left or right
- Goal: maximize total reward

Note:

- The agent knows nothing more about the environment (no physics, no experience, ...)
- We might hand pick a policy: $\pi(a|s) = \leftarrow$ if $\theta > \pi/2$... but it won't work well
- Learning to balance the pole requires understanding long(ish) range consequences
- To explore new possibilities, agent must sometimes try unlikely (in π) actions...

EXAMPLE: MS PACMAN





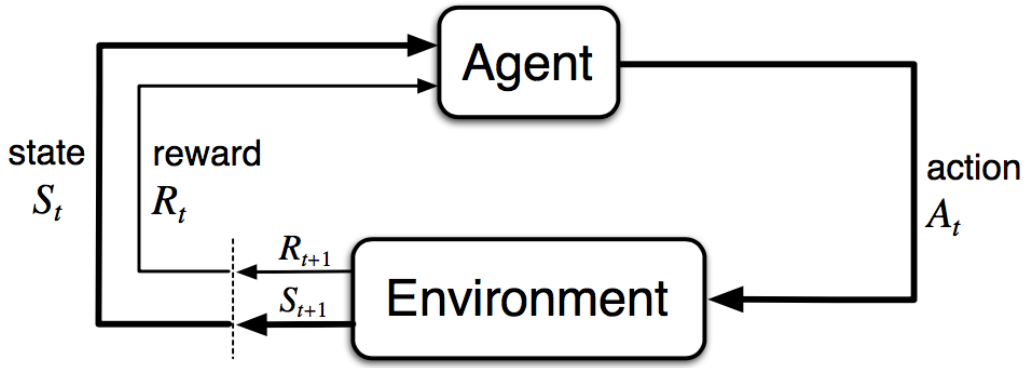
Example choices:

- State $S_t = [x_t^p, d_t^p, x_t^{g0}, d_t^{g0}, ...]$, position/direction of you, ghosts, pips, fruits, etc...
- Reward $R_t \in \{+10, +100, -1000, 0\}$ for pip, ghost, loss of life, doing nothing
- Action $A_t = \{\leftarrow, \rightarrow, \uparrow, \downarrow\}$; move Ms PacMan
- Goal: maximize total reward

Note:

- Many possible states can result in different problem difficulty
- Rewards can also be designed/mapped to features: score board vs loss of life (if multiple)
- How to balance short term rewards (pips) with big wins (ghosts, not dying,...)?
- How might you balance being greedy about things you know, vs learning new things?

EXAMPLE: NOT JUST AI



Example choices:

- State S_t = food availability/freshness, appetite, dishes, etc...
- Reward R_t involves speed, quality, spills, expense, satiety, hunger, etc...
- Action A_t of multiple steps involving delay/planning/experience.
- Goal: maximize total reward

Note:

- When/how often should you try cooking something new?
- When/how often should you do what you know works?

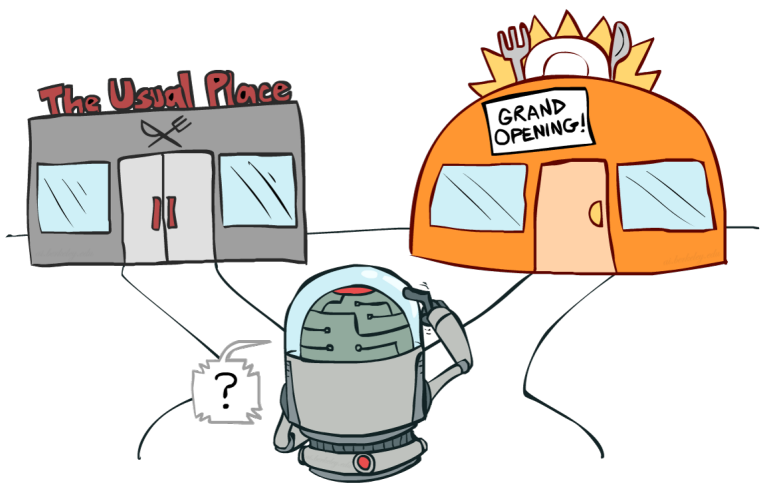
EXPLORATION / EXPLOITATION

The previous examples point to *exploration / exploitation*

- a fundamental concept in reinforcement learning
- recall that operating under uncertainty is fundamental to RL
- Any notion of "best action" is really only "best given what I know so far"
- A random or believed-suboptimal action may underperform, but it should teach us something

A fundamental tradeoff/conflict

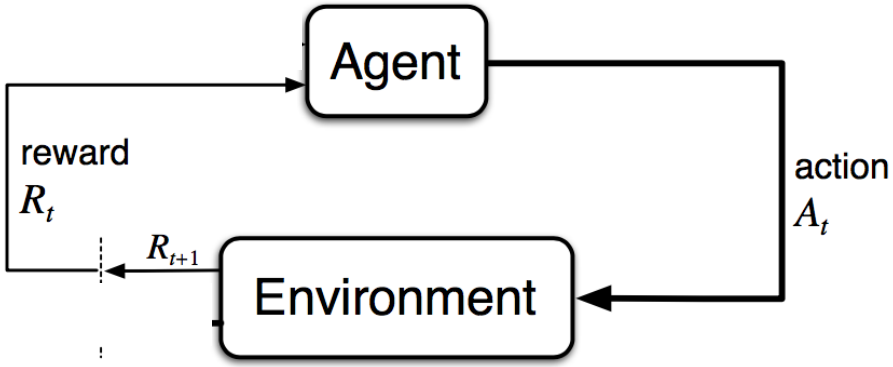
- Exploitation accrues more near-term reward, but learns little new
- Exploration sacrifices short-term reward, but accrues information



[from http://slides.com/ericmoura]

THE MULTI-ARMED BANDIT PROBLEM

To elucidate the exploration/exploitation tradeoff, we consider the *multi-armed bandit* problem



You face K slot machines (used to be called *one-armed bandits*, hence...)

- You choose which slot machine to play: action $a_t = k$
- Rewards payoff with parameter $\mu_1, ..., \mu_K$; eg: $r_t|a_t = k \sim Bern(\mu_k)$ or $\sim \mathcal{N}(\mu_k, 1)$
- The probabilities are *unknown*; you must discover them through your action sequence
- This is a fixed-state (or nonassociative) RL problem: actions don't change environment
- You have to find the best machine, and play it enough to accrue max reward
- (not just a thought experiment: think A/B testing on an ecommerce site)

Note there is a big literature on bandits: different rewards, dueling bandits, contextual bandits, adversarial bandits, etc. Here we deal only with the simplest case.

DEFINITIONS

We are interested in accruing maximum reward. Some important definitions:

- Define reward r_t and action a_t as previous.
- Define *value function* $q(a_t = k) = E(r_t | a_t = k)$: expected reward for playing machine k
- Define optimal sequence (theoretical, not achievable) as $\max_a q(a)$
- We then equivalently attempt to minimize *regret*:

$$L(T) = \sum_{t=1}^{T} \max_a q(a) - E\left(\sum_{t=1}^{T} q(a_t)\right)$$

- ...how much we regret our sequence of actions, if we later learned the best choice.

Strategies:

- *Greedy*: only exploit, pick $\mu_{\hat{k}}$, what you believe to be the best so far

$$L(T) = T\left(\max_k \mu_k - \mu_{\hat{k}}\right) \qquad \text{linear regret in } T!$$

- Random: only explore, pick at random

$$L(T) = T\left(\max_k \mu_k - \frac{1}{K}\left(\mu_1 + ... + \mu_K\right)\right) \qquad \text{linear regret in } T!$$

We hope to achieve *sublinear regret* with more sensible policies $\pi(a)$

SIMPLE BALANCE OF EXPLORATION AND EXPLOITATION

ϵ-greedy policy is a simple mixture of greedy and random:

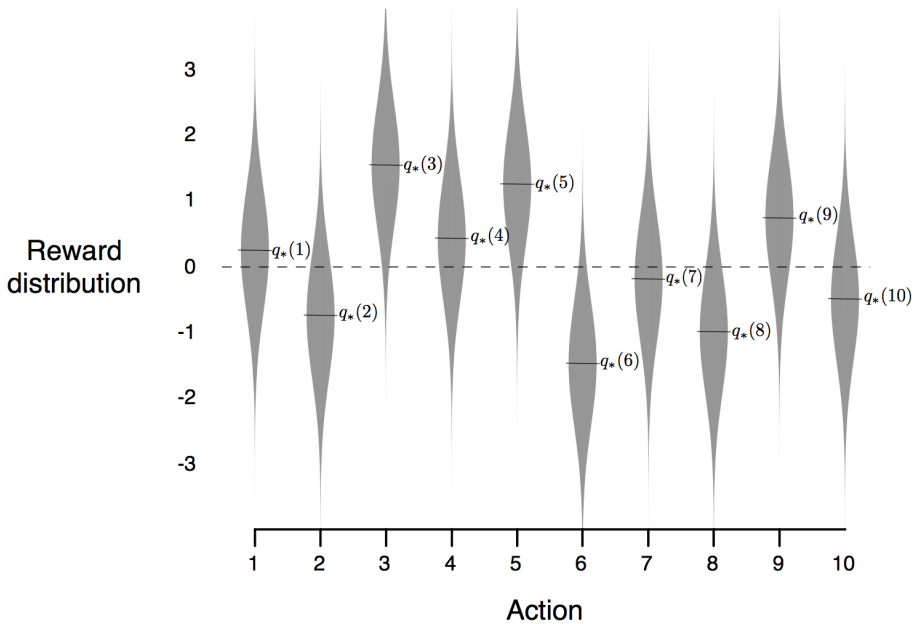
$$a_t = \begin{cases} \arg\max_k Q(a_t = k) & \text{with probability } 1 - \epsilon \\ k \sim Unif(1, ..., K) & \text{with probability } \epsilon \end{cases}$$

Value function $Q(a_t)$ estimates true action value $q(a_t = k) = E(r_t | a_t = k)$. Update:

$$Q(a_t = k) \leftarrow Q(a_t = k) + \frac{1}{n_k}\left(r_t - Q(a_t = k)\right) \quad \text{or equivalently, } \leftarrow \frac{1}{n_k}\sum_{i=1}^{n_k} r_i$$

learning the action-value function q...

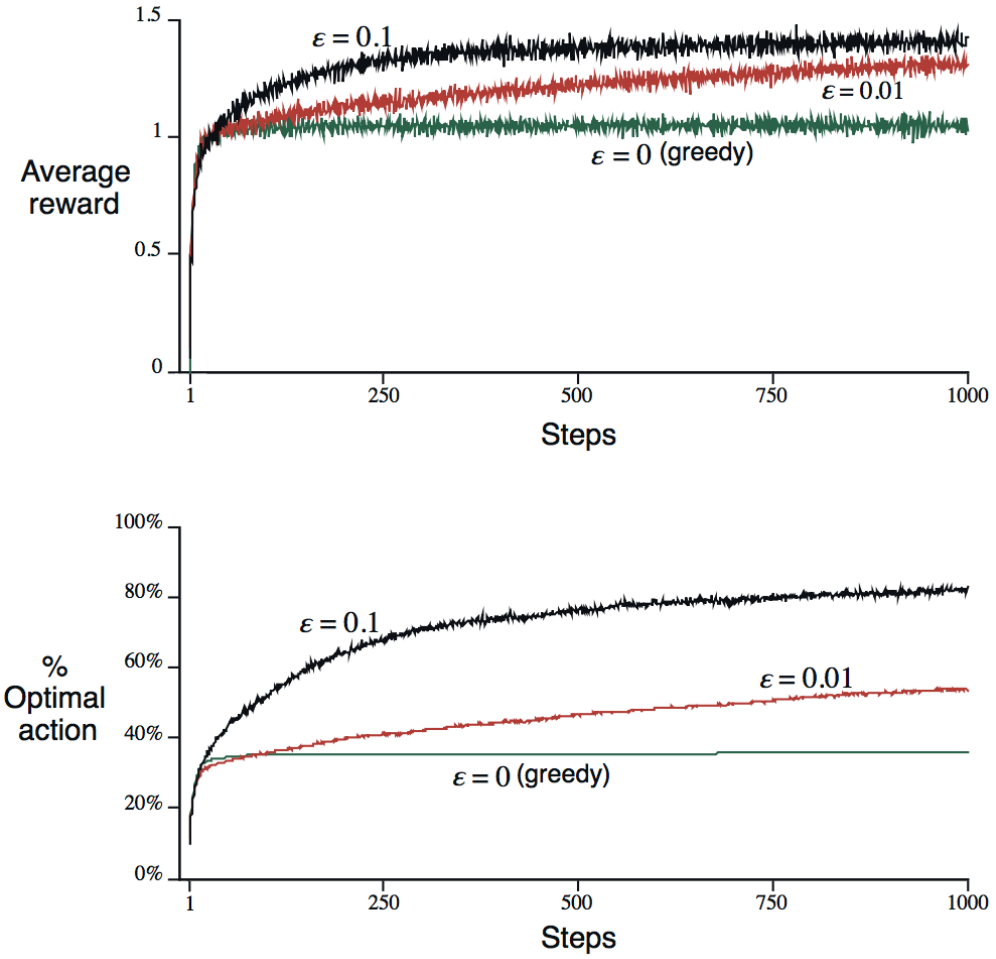
Consider the following 10-armed case with Gaussian bandits $r_t | a_t = k \sim \mathcal{N}(\mu_k, 1)$



[Sutton and Barto... note they use $q_*(k)$, not μ_k]

PERFORMANCE OF ϵ-GREEDY

Performance improves over greedy approach...



[Sutton and Barto]

Better, but still linear regret (and ϵ depends on uncertainty/variance in the problem)

Upper Confidence Bound

ϵ-greedy exploits well ($1 - \epsilon$ of the time), but explores randomly. Suppose instead:

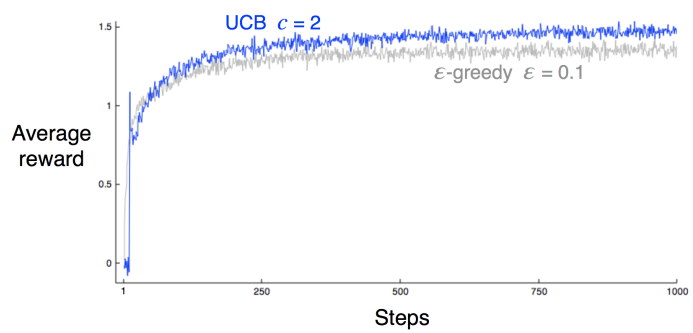
- we maintain a confidence interval on each μ_k
- we already have "posterior" mean $Q(a_t = k)$; define confidence interval as $\sigma^2(a_t = k)$
- Explore arms where there is reasonable probability of a higher value
- Using our posterior belief, we select (for some constant c)

$$k^* = \arg\max_k \left(Q(a_t = k) + c\sigma(a_t = k)\right)$$

- if t is large, confidence should be high $\rightarrow$ greedy exploitation
- if t is small, confidence low $\rightarrow$ exploration
- We call such methods *UCB*, and they require an estimate of confidence. Good choice:

$$k^* = \arg\max_k \left(Q(a_t = k) + c\sqrt{\frac{\log t}{n_k}}\right)$$

UCB achieves $\log(T)$ regret, there is a great deal known about it theoretically, and it generalizes well



[Sutton and Barto]

THOMPSON SAMPLING

Consider the K Bernoulli bandits problem:

- $r_t|a_t = k \sim Bern(\mu_k)$
- Setup otherwise identical to previous. Recall Bayesian modeling and conjugacy
  - Our prior (uninformed/uniform) belief is $\mu_k \sim Beta(1, 1)$. Recall:

$$p(\mu) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \mu^{\alpha-1}(1-\mu)^{\beta-1}$$

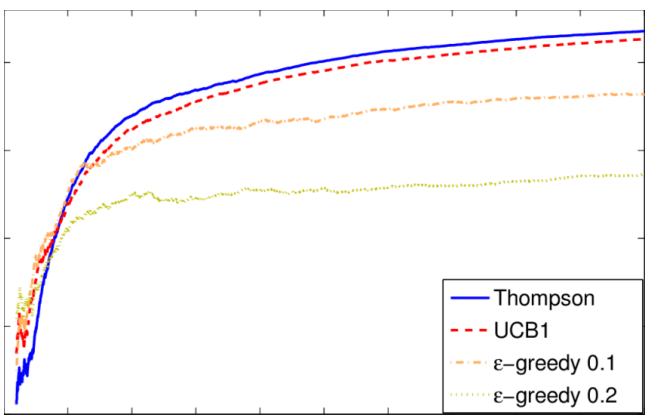
  - Each observation updates beliefs easily with Beta-Bernoulli conjugacy:

$$\mu_k|n_k^0, n_k^1 \sim Beta\left(1 + n_k^1, 1 + n_k^0\right)$$

Thompson sampling:

- Initialize all arms with $\mu_k \sim Beta(1, 1)$
- At time t, sample $s_t(k) \sim Beta\left(1 + n_k^1, 1 + n_k^0\right)$
- Play $k^* = \arg\max_k s_t(k)$
- Update n_{k+1}^1, n_{k+1}^0 based on r_t

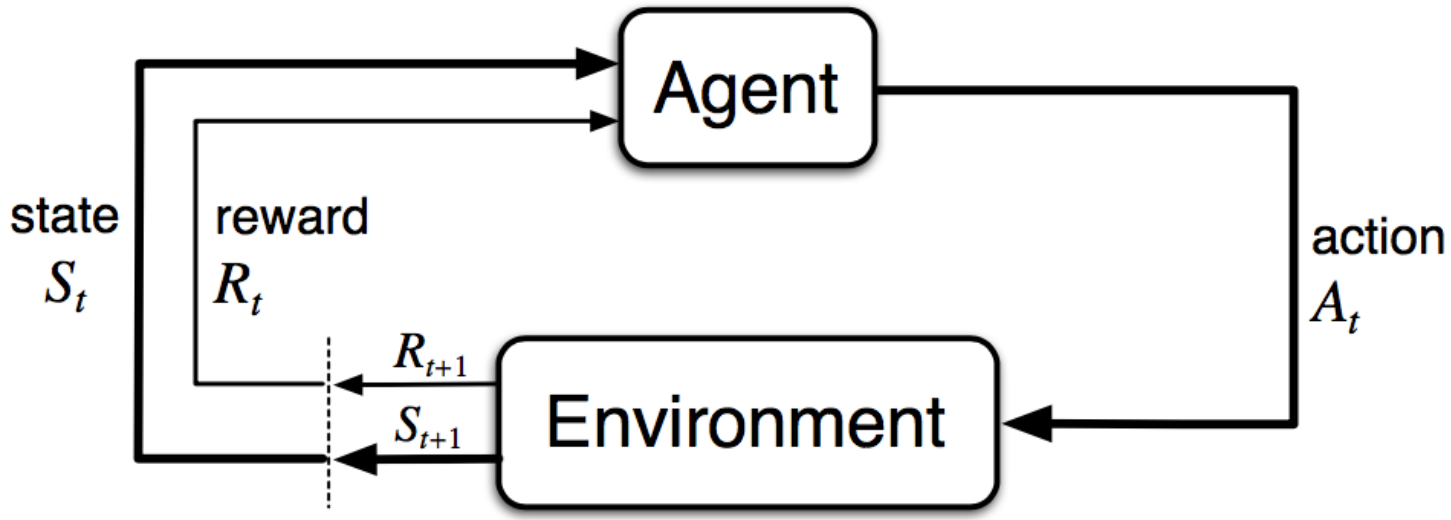
Thompson sampling achieves $\log(T)$ regret, outperforms many methods in practice, and generalizes to several settings



[Xu et al 2013; reward % vs time]

RECAP

Reinforcement Learning is the study of problems that can be characterized by...



- Actions: agent takes *action* A_t at time t
- Rewards: agent/environment receives/produces *reward* R_t
- State: environment updates *state* S_t (fixed in multi-armed bandit)

We choose/learn/design a policy $\pi(a|s)$, such as (in the bandit problem):

$$a_t = \begin{cases} \arg\max_k Q(a_t = k) & \text{with probability } 1 - \epsilon \\ k \sim Unif(1, \ldots, K) & \text{with probability } \epsilon \end{cases}$$

Recall (in the bandit case) the action-value function $Q(a_t = k) \approx q(a_t = k) = E(r_t|a_t = k)$

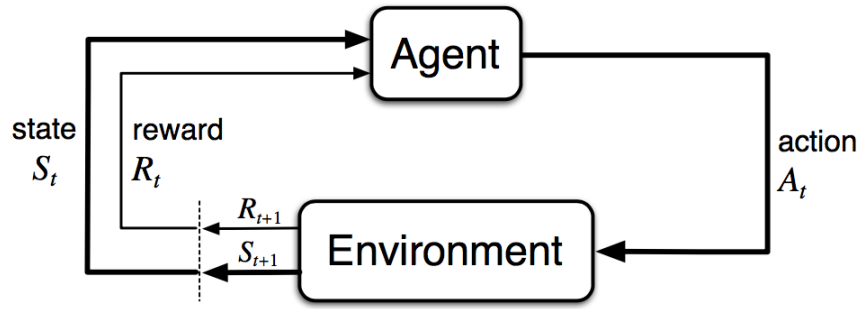
From Bandits to Markov Decision Process

What if the state changes based on our actions?

- Now our experience flows as:

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \ldots$$

- and the reward distribution (and *action-value* function...) should now depend on state



A *Markov Decision Process* (MDP) is defined by:

$$p(s', r|s, a) \triangleq p\left(S_{t+1} = s', R_t = r|S_t = s, A_t = a\right)$$

Recall Markov property $p(S_t|S_{t-1}, \ldots, S_1) = p(S_t|S_{t-1})$

- future and past are conditionally independent, given the present.
- If I tell you where I am now, the history of how I got here is irrelevant.
- Markovity is not a "without loss of generality statement"... we are simplifying/approximating (but r-Markov can mitigate)
- MDP can be viewed as a collection of action-switched Markov chains on states (a tensor)

The MDP equation (in discrete setting, for clarity):

$$p(s', r|s, a) \triangleq P\left(S_{t+1} = s', R_t = r|S_t = s, A_t = a\right)$$

- State transition probabilities

$$p(s'|s, a) = P\left(S_{t+1} = s'|S_t = s, A_t = a\right) = \sum_r p(s', r|s, a)$$

...marginalizing over reward distribution

- Reward expectation

$$r(s, a) \triangleq E\left(R_t|S_t = s, A_t = a\right) = \sum_r r \sum_{s'} p(s', r|s, a)$$

...marginalizing over destination state, expecting over reward

- And more...

MDPs offer a highly successful framework for many reinforcement learning problems.

GOALS, RETURNS, EPISODES

Desire to "maximize reward" now needs more detail... we define *return* G_t:

- $G_t \triangleq R_t + R_{t+1} + \ldots + R_T$ or $G_t \triangleq R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots$
- *Discount factor* γ prioritizes near term rewards
- Generally: $G_t = \sum_{k=0}^{T} \gamma^k R_{t+k}$ and note: $G_t = R_t + \gamma G_{t+1}$

Now we can define the central functions that help us understand the value of states and actions:

- *state-value function* for all states s:

$$v_\pi(s) \triangleq E_\pi\left(G_t | S_t = s\right) = E_\pi\left(\sum_{k=0}^{T} \gamma^k R_{t+k} | S_t = s\right)$$

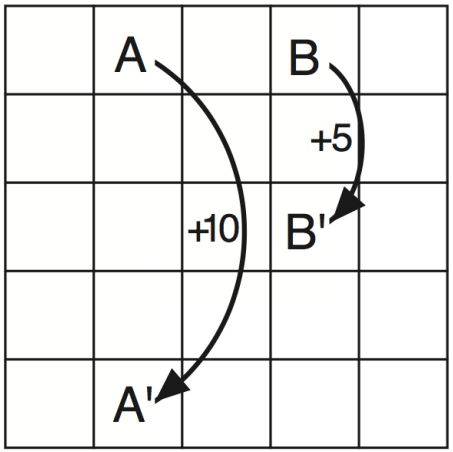
- *action-value function* for all state-action pairs (s, a)

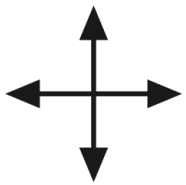
$$q_\pi(s,a) \triangleq E_\pi\left(G_t | S_t = s, A_t = a\right) = E_\pi\left(\sum_{k=0}^{T} \gamma^k R_{t+k} | S_t = s, A_t = a\right)$$

Note

- value functions depend on a policy π
- Better policies increase value...

EXAMPLE: GRIDWORLD





Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

$$S_t = \begin{cases} S_{t-1} & \text{if } A_t \text{ would leave grid} \\ A' & \text{if } S_{t-1} == A \\ B' & \text{if } S_{t-1} == B \\ S_{t-1} + A_t & else \end{cases} \qquad R_t = \begin{cases} -1.0 & \text{if } A_t \text{ would leave grid} \\ +10.0 & \text{if } S_{t-1} == A \\ +5.0 & \text{if } S_{t-1} == B \\ 0.0 & else \end{cases}$$

Consider a random policy $\pi(a|s) = Unif(\leftarrow, \rightarrow, \uparrow, \downarrow)$, with discount $\gamma = 0.9$

$$v_\pi(s = A) = 10.0 + 0.9\left(\frac{1}{4}(-1.0 + 0.9(...)) + \frac{1}{4}(0.0 + 0.9(...)) + ...\right)$$

- The *Bellman equation* recursively defines the value function:

$$v_\pi(s) = \sum_{a,s',r} \pi(a|s)p(s', r|s, a)\left(r + \gamma v_\pi(s')\right)$$

- Bellman equations are central to RL but not entirely needed for our purposes.
- Takeaway: $v_\pi(s)$ is a solution to some linear equations

POLICY ITERATION

Key conceptual points:

- A policy π induces value functions $v_\pi(s)$ and $q_\pi(s,a)$
- The value functions capture our expected return $\rightarrow$ the objective of the RL problem
- Tools like the Bellman equation (in some settings) let us calculate the value functions
- Improving the policy should increase value...

Consider the action-value function:

$$q_\pi(s,a) = E_\pi\left(\sum_{k=0}^{T}\gamma^k R_{t+k}|S_t = s, A_t = a\right) = E_\pi\left(R_t + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\right)$$

The *policy improvement theorem* says:

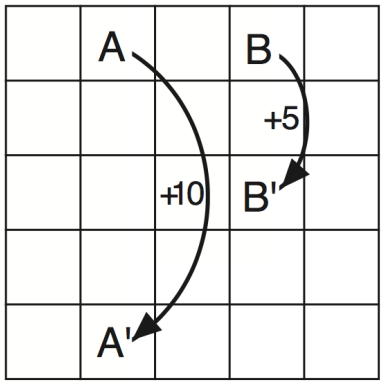
- for deterministic policies $\pi : \mathcal{S} \rightarrow \mathcal{A}$ (simpler than the usual $\pi(a|s)$)
- if $\exists\pi'$, π such that for all s
$$q(s,\pi'(s)) \geq v_\pi(s)$$
- then π' is a better policy than π in the sense that:
$$v_{\pi'}(s) \geq v_\pi(s) \;\; \forall s$$

Notice that a greedy policy $\pi'(s) = \arg\max_a q_\pi(s,a)$ by definition will satisfy! Thus:

- Policy improvement is reasonably straightforward (greedy, ϵ-greedy,...)
- Policy evaluation (calculating $v_\pi(s)$) is necessary in this framework
- Iterating between these two is *policy iteration*

OPTIMALITY

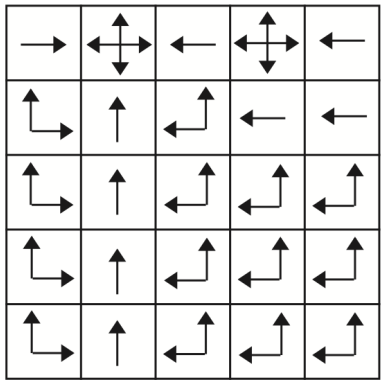
Policy iteration will result in an optimal value function v_*



Gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

- The value functions will satisfy *Bellman optimality*

$$v_*(s) = \max_a q_*(s,a) = \max_a \sum_{s',r} p(s',r|s,a)\left(r + \gamma v_*(s')\right)$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left(r + \gamma \max_{a'} q_*(s',a')\right)$$

- Extracting π^* from v_*: search over states, choose action to get there.
- Extracting π^* from $q_*(s,a)$: search over actions, choose max.
- The point: we have means to increase value via our policy $\rightarrow$ solving RL

Unfortunately, calculating v_π is only possible in simplistic (known) cases, so much work to do...

LEARNING $v_\pi(s)$ WITH TEMPORAL DIFFERENCES

Temporal Difference (TD) learning

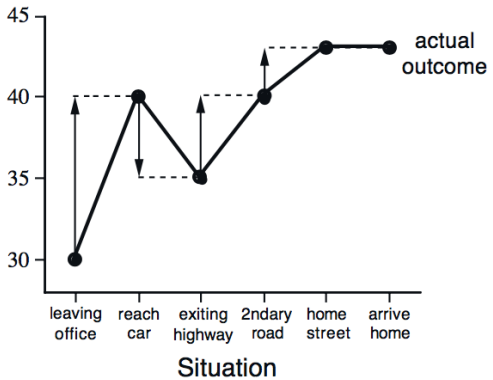
- We seek to learn $v_\pi(s)$ in an online fashion while acting according to policy $\pi(a|s)$
- Suppose we have an estimator $V_\pi(s)$ of the value function
- Define the *TD error* as the difference between what you received/anticipated:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

- Update your estimate with that error signal (and step size α):

$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t$$

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43



TD learning on v_π:

- is *prediction* without a model: give me a policy and I'll tell you its value
- provably convergent (if α is correctly scheduled...)
- fully online, bootstraps estimates from estimates (V)

LEARNING $q_\pi(s,a)$ WITH TEMPORAL DIFFERENCES

With a given policy $\pi(a|s)$, TD learning can be directly applied to learning the action-value function

$$\begin{aligned} \delta_t &= r_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t) \\ Q_\pi(s_t, a_t) &\leftarrow Q_\pi(s_t, a_t) + \alpha\delta_t \end{aligned}$$

For each update we need $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. (SARSA)

- enjoys same online/bootstrapping behavior of all TD methods
- *on-policy*: chooses actions from one policy and learns from the same policy

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

[Sutton and Barto]

On-policy TD learning is a compromise:

- ideally, the *learned* policy is optimal and greedy
- but it must *behave* suboptimally to adequately explore

Q-LEARNING

Key idea: use two policies and learn from *off-policy* actions

- maintain an ϵ-greedy *behavior policy* to choose actions
- learn (update Q) according to a greedy policy

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

[Sutton and Barto]

Note key difference between Q-learning and SARSA:

- SARSA chooses a_{t+1} (A') from the ϵ-greedy policy
- Q-learning updates $Q(s,a)$ with the greedy $\max_a Q(s_{t+1}, a)$

Remember: once q is learned, the optimal policy is simple (intuition: cf. dropout, reg.,...)

$$\pi(a|s) = \begin{cases} 1 & a = \arg\max_{a'} q(s,a') \\ 0 & else \end{cases}$$

Q-LEARNING

Off-policy TD control (Q-learning) was a major breakthrough in RL

- learns optimal policy while following an exploratory policy
- enables observation of humans (expert/coach imitation) or other agents
- allows reuse of data generated from old policies

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

[Sutton and Barto]

The problem (with most interesting RL settings):

- the state/action space is too big $\rightarrow$ impractical to sample entirely
- the state/action space is continuous $\rightarrow$ impossible to sample entirely
- How to scale up Q-learning?

Idea: approximate $q(s,a)$ with a parameterized function $Q_\theta(s,a)$...

CartPole and OpenAI gym

We will use the handy *gym* environment from OpenAI

```
# install OpenAI gym per https://gym.openai.com/docs/
import gym
import numpy as np
import matplotlib.pyplot as plt
env = gym.make('CartPole-v0')
```

```
[2017-11-21 14:11:55,562] Making new env: CartPole-v0
```

Important to understand the data

```
# what are the observations and actions?
print(env.observation_space)
print(env.observation_space.low)
print(env.observation_space.high)
print(env.action_space)
# Discuss comment from gym docs: "Fortunately, the better your learning algorithm,
# the less you'll have to try to interpret these numbers yourself."
# see also https://github.com/openai/gym/wiki/CartPole-v0 and note errors
```

```
Box(4,)
[ -4.80000000e+00  -3.40282347e+38  -4.18879020e-01  -3.40282347e+38]
[  4.80000000e+00   3.40282347e+38   4.18879020e-01   3.40282347e+38]
Discrete(2)
```

Use available docs/wikis/blogs, but be careful...

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

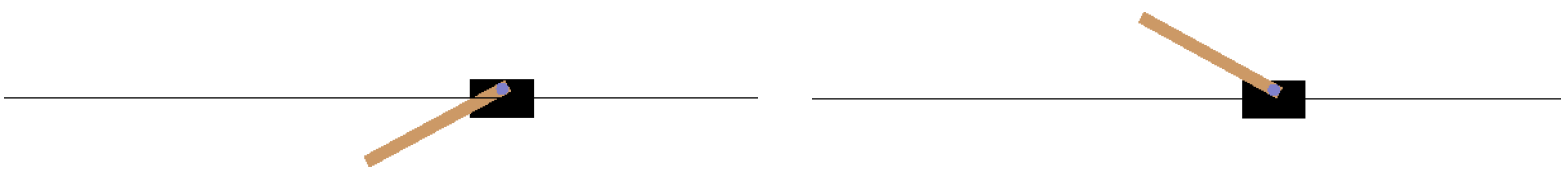
OpenAI gym

Random actions...

```python
# See https://github.com/openai/gym/wiki/CartPole-v0
# Observation: [cart pos, cart vel, pole angle, pole vel]
# Note: bad docs... observation[2] is denominated in radians, so 'done' at +-0.21
# we will run some episodes to watch it fail
for ep in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        # randomly sample an action
        action = env.action_space.sample()
        # take the action, and the environment responds
        observation, reward, done, info = env.step(action)
        print('step {}, action {}, reward {}, observation {}'.format(t,action,reward*(not done),observation))
```

```
step 0, action 0, reward 1.0, observation [ 0.00110925 -0.21506057  0.00975978  0.32138836]
step 1, action 1, reward 1.0, observation [-0.00319196 -0.02007896  0.01618755  0.0317992 ]
step 2, action 1, reward 1.0, observation [-0.00359354  0.17480716  0.01682353 -0.25573275]
```

Rendering



AGENT CLASS

To organize our thinking, we create an Agent class

```
class Agent:

    def __init__(self, policy='random'):
        # first what reward has the agent accrued so far (we would call this return, but...)
        self.total_reward = 0
        self.policy = policy

    def choose_action(self, observation):
        # act according to the policy
        if self.policy=='random':
            return int(np.round(np.random.random()))
        elif self.policy=='left_right':
            if observation[2]>0.0:
                return 1
            else:
                return 0

    def gather_reward(self, reward):
        self.total_reward += reward
    def get_total_reward(self):
         return self.total_reward
    def set_total_reward(self, new_total):
         self.total_reward = new_total
```

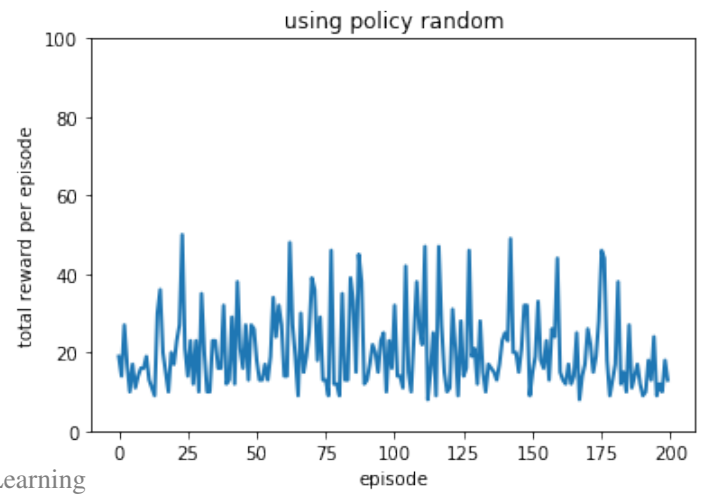
Agent contains a (fixed) policy and acts according to $\pi(a|s)$.

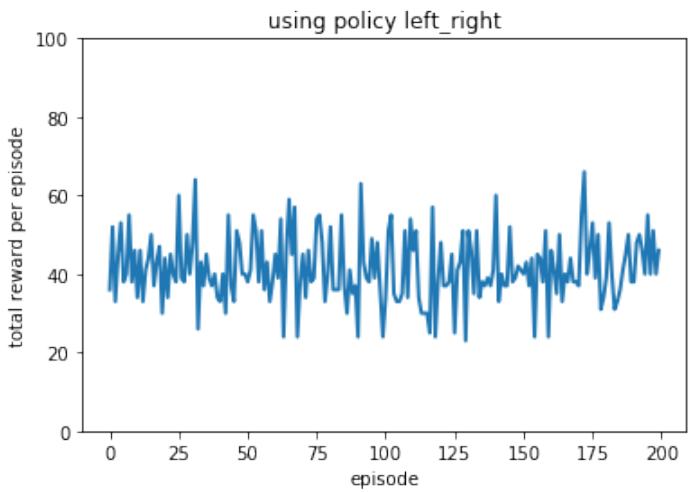
PERFORMANCE OF NAIVE POLICIES

CartPole episode ends when the pole is ± 0.2 radians away from center (or $T = 200$)

```
policy = 'left_right'
agent = Agent(policy)
ep_rewards = []
for ep in range(20):
    # reset environment and agent reward
    last_observation = env.reset()
    agent.set_total_reward(0)
    for t in range(100):
        # note that rendering is the vast amount of the computational effort.  Disable once comfortable...
        #env.render()
        # randomly sample an action
        action = agent.choose_action(last_observation)
        # take the action, and the environment responds
        observation, reward, done, info = env.step(action)
        # update agent
        agent.gather_reward(reward)
        last_observation = observation
        if done==True:
            print('Episode {} died after time {} with total reward {}'.format(ep, t, agent.get_total_reward()))
            ep_rewards.append(t)
            break
    if (ep+1) % 20 == 0:
        print('Average total reward so far {}'.format(np.mean(ep_rewards)))
```

Simple fixed policies don't learn and do rather poorly ($\sim 40 \ll 200$)





ELABORATING THE AGENT CLASS

We will approximate $Q(s,a) \approx Q_\theta(s,a)$

- In the simplest case: $Q_\theta(s,a) = \theta_{\partial s, \partial a}$, where ∂s denotes a discretized index of s.

```
32      def obs_index(self, observation):
33          # a helper method to discretize the observation
34          bins = (np.array([1e20]),
35                  np.array([1e20]),
36                  np.array([-0.2,0,0.2]),
37                  np.array([-.3,.3])
38                  )
39          ind=np.zeros(4).astype(int)
40          for i in range(len(observation)):
41              ind[i] = np.digitize(observation[i],bins[i])
42          return tuple(ind)
43
44      def q(self, observation):
45          # now return the q function value for both actions
46          ind = self.obs_index(observation)
47          return self.theta[ind]
```

- To use Q learning we will need more parameters in our class:

```
3       def __init__(self, policy='random'):
4           # first what reward has the agent accrued so far (we would call this return, but...)
5           self.total_reward = 0
6           self.policy = policy
7           # discount, learning, exploration rates
8           self.gamma = 0.99
9           self.alpha = 1.0
10          self.epsilon = 0.2
11          # we will make q a nonparametric lookup table over q(s0,s1,s2,s3,a)
12          # s is continuous so we will discretize for simplicity
13          self.theta = np.zeros([1,1,4,3,2])
```

Q-LEARNING IN PRACTICE

Suppose Q is learned; we behave according to the ϵ-greedy policy induced by Q:

```
57      def choose_action(self, observation):
58          # act according to the behavior policy
59          if self.policy=='random':
60              return int(np.round(np.random.random()))
61          elif self.policy=='left_right':
62              if observation[2]>0.0:
63                  return 1
64              else:
65                  return 0
66          elif self.policy=='q_discretized':
67              # an epsilon greedy policy
68              if np.random.rand() > self.epsilon:
69                  if self.q(observation)[0]>self.q(observation)[1]:
70                      return 0
71                  else:
72                      return 1
73              else:
74                  # explore
75                  return int(np.round(np.random.random()))
```

Questions:

- How is the ϵ-greedy policy actuated here?
- Where does the greedy choice take place?
- What does the object `self.q(observation)` represent?

Q-LEARNING IN PRACTICE

Reminder: the Q-learning algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

In code:

```
for ep in range(1001):
    # reset environment and agent
    last_observation = env.reset()
    agent.set_total_reward(0)
    # done at T==199 so no reason to go further
    for t in range(201):
        # agent chooses an action
        action = agent.choose_action(last_observation)
        # agent takes the action, and the environment responds
        observation, reward, done, info = env.step(action)
        # update agent with reward
        agent.gather_reward(reward)
        # update q function based on result
        agent.q_update(last_observation,action,reward,observation)
        # iterate
        last_observation = observation
        if done==True:
            ep_rewards.append(agent.get_total_reward())
            break
```

Q-LEARNING IN PRACTICE

Reminder: the Q-learning algorithm

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

That core Q update:

```
49      def q_update(self, last_observation, action, reward, observation):
50          # core Q-learning step
51          ind = self.obs_index(observation)
52          ind_last = self.obs_index(last_observation)
53          # NOTICE: This is a different implicit learning policy than the behavior policy
54          delta = (reward + self.gamma*np.max(self.theta[ind]) - self.theta[ind_last+(action,)])
55          self.theta[ind_last+(action,)] += self.alpha*delta
```

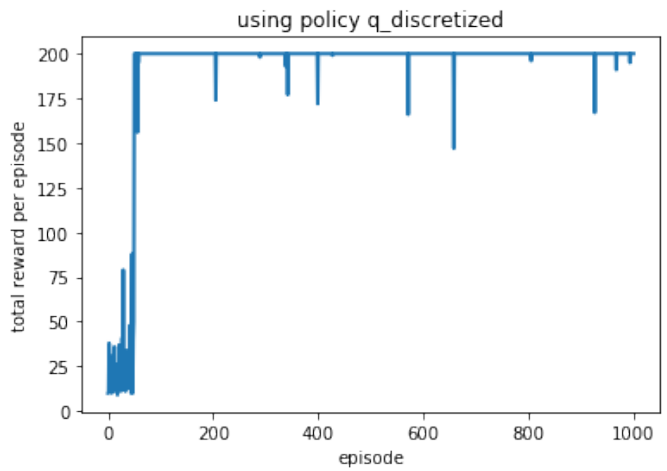
Note the difference between the behavior policy and the learned policy; this is *off-policy*

PERFORMANCE OF SIMPLE Q-LEARNER

Reminder: Q-learning in code

```
for ep in range(1001):
    # reset environment and agent
    last_observation = env.reset()
    agent.set_total_reward(0)
    # done at T==199 so no reason to go further
    for t in range(201):
        # agent chooses an action
        action = agent.choose_action(last_observation)
        # agent takes the action, and the environment responds
        observation, reward, done, info = env.step(action)
        # update agent with reward
        agent.gather_reward(reward)
        # update q function based on result
        agent.q_update(last_observation,action,reward,observation)
        # iterate
        last_observation = observation
        if done==True:
            ep_rewards.append(agent.get_total_reward())
            break
```

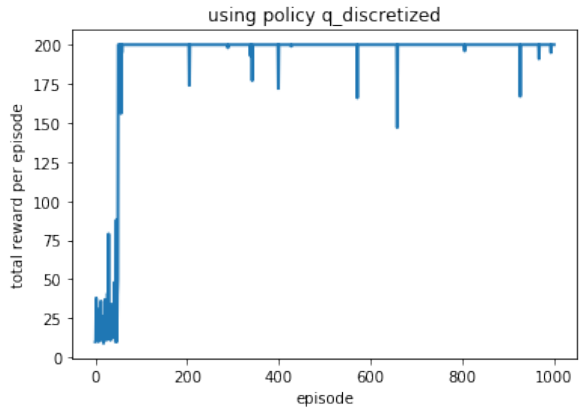
Vastly improved performance



Note some dips in performance in some episodes... why?

REMINDER: WHEN ϵ-GREEDY IS USEFUL

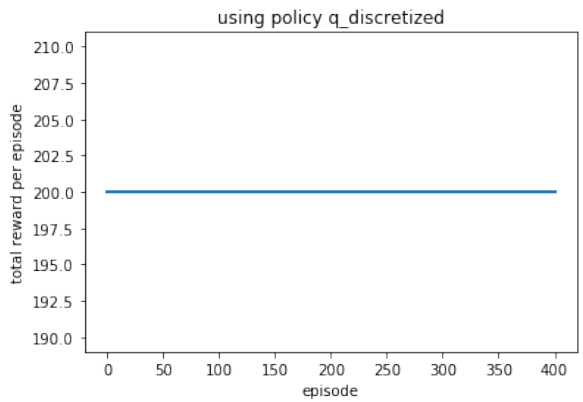
We used an ϵ-greedy behavior policy to *explore*



Once we have learned $q(s, a)$, we now only want to *exploit*. Control without learning:

```
9     for t in range(201):
10        #env.render()
11        action = agent.choose_action(last_observation)
12        observation, reward, done, info = env.step(action)
13        agent.gather_reward(reward)
14        last_observation = observation
```

Greedy performance (render and watch the learned policy)



INTERROGATE $Q_\theta(s, a)$

Inspecting the learned Q function

- ...clarifies what the Q function really is
- ...develops intuition for how the control agent performs
- ...sanity checks what the algorithm has learned

```
# look at the q function to really understand what it is doing...
print(agent.theta[0,0,:,:,0])
print('')
print(agent.theta[0,0,:,:,1])
# Recall
#bins = (np.array([1e20]),
#                np.array([1e20]),
#                np.array([-0.2,0,0.2]),
#                np.array([-.3,.3])
#                )
# and 0== LEFT , 1==RIGHT
```

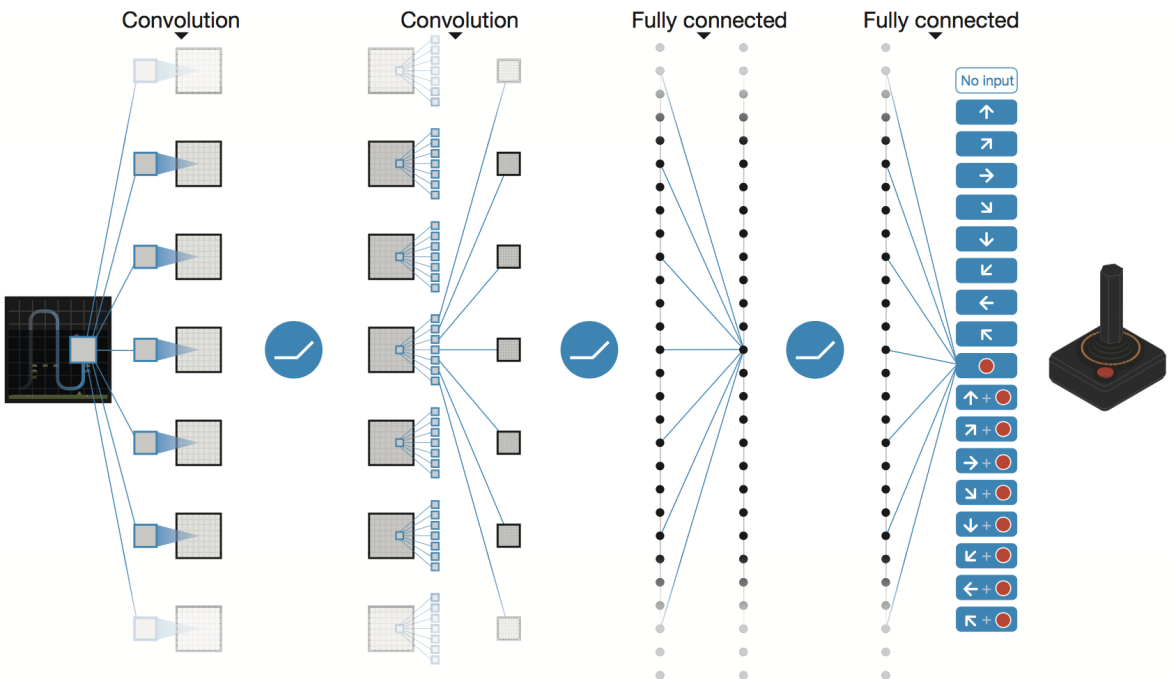
```
[[  1.48386329   2.43854066   0.        ]
 [100.         100.          99.99491416]
 [100.         100.          98.33991503]
 [  0.           0.           0.29229841]]

[[  2.10196704   0.53150942   7.53990426]
 [ 99.76560516 100.         100.        ]
 [ 99.99832557 100.         100.        ]
 [  0.           0.           1.45032334]]
```

From Q tables to Deep Q Networks

The simple tabular function:

- is easy to learn
- has only a few parameters
- can not share information across states
- can not scale up to large state spaces or large action spaces

Idea: make $Q_\theta(s, a)$ a deep network



[Mnih et al (2015)]

DEEP Q NETWORKS

Key enabling idea: maintain a memory of data to use as *experience replay*

```python
class Replay:
    # accepts a tuple (s,a,r,s') and keeps a list, returns a random batch of tuples as needed
    # remember this is q learning, so a' is not needed (why?... off policy argmax_a)
    def __init__(self):
        self.buffer = []
        self.length = 0
        self.max_length = 100000

    def write(self, data):
        if self.length >= self.max_length:
            # drop oldest data point to make room for new
            self.buffer.pop(0)
            self.length -= 1
        self.buffer.append(data)
        self.length += 1

    def read(self, batch_size):
        # randomly sample a batch and return a list thereof
        return random.sample(self.buffer,min(batch_size,self.length))

```

Because Q-learning is off policy, this buffer enables us:

- to explore with an ϵ-greedy behavior policy, gathering plenty of experience data
- use those experiences to *replay* a mini-batch (s_i, a_i, r_i, s_{i+1})
- train a network in our usual supervised fashion, with the objective:

$$\min_{\theta} \quad (y_i - Q_\theta(s_i, a_i))^2$$

$$\text{where} \quad y_i = \begin{cases} r_i & \text{if } s_{t+1} \text{ is terminal} \\ r_i + \gamma \max_a Q_{\theta^{old}}(s_{i+1}, a) & \text{else} \end{cases}$$

- Note: θ^{old} simply indicates that y_i is a fixed target for training (0 gradient wrt θ).

BUILD A NETWORK

Now we need a network $Q_\theta(s, a)$

- it will take as input a state $s_t \in \mathbb{R}^4$
- it will return as output a vector $\begin{bmatrix} Q_\theta(s_t, a_t = 0) \\ Q_\theta(s_t, a_t = 1) \end{bmatrix}$
- it will be a regression network (ie not the usual softmax as in CNN)

```python
class Network:

    def __init__(self, session, n_in , n_out):
        self.session = session
        self.n_in = n_in
        self.n_out = n_out
        self.n_hidden = 60
        # data placeholders
        self.x = tf.placeholder(tf.float32, [None, n_in], name='x')
        self.y = tf.placeholder(tf.float32, [None, n_out], name='y')
        self.x_in = tf.reshape(self.x, [-1,self.n_in])
        # 2 layer network
        self.W_fc1 = tf.get_variable('W_fc1', shape=[self.n_in,self.n_hidden])
        self.b_fc1 = tf.get_variable('b_fc1', shape=[self.n_hidden])
        self.h_fc1 = tf.nn.relu(tf.add(tf.matmul(self.x_in, self.W_fc1), self.b_fc1, name='layer1'))
        self.W_fc2 = tf.get_variable('W_fc2', shape=[self.n_hidden,self.n_out])
        self.b_fc2 = tf.get_variable('b_fc2', shape=[self.n_out])
        self.q = tf.add(tf.matmul(self.h_fc1, self.W_fc2), self.b_fc2, name='layer2')
        # loss, train_step, etc.
        self.loss = tf.reduce_sum(tf.square(self.y - self.q),1)
        self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.loss)

    def compute(self, x):
        # evaluate the network and return the action values [q(s,a=0),q(s,a=1)]
        return self.session.run(self.q, feed_dict={self.x:np.reshape(x,[-1,self.n_in])})

    def train(self, x_batch, y_batch):
        # take a training step
        _ = self.session.run(self.train_step, feed_dict={self.x: x_batch, self.y: y_batch})
```

REMINDER: TAKE INCREMENTAL STEPS

Caution:

- it is easy to get lost between Q-learning, the network, tensorflow, etc.
- make good design choices (eg abstract as much `tf` as possible into Network class)
- test the network before involving the Q-learning complexity

```
# simple demonstration that network is able to train properly.
# allows us to confirm network function before putting it in the RL problem...
with tf.Graph().as_default():

    with tf.Session() as sess:

        f = Network(sess, 4, 2)

        # usual tf initialization
        sess.run(tf.global_variables_initializer())

        x = np.random.randn(10000,4)
        # some silly function that I hope a 2 layer network could (roughly) learn
        y = np.transpose([ x[:,0]+x[:,1]**2, x[:,2]+x[:,3]**3 ])

        print('MSE at iteration 0 is {}'.format(((f.compute(x) - y)**2).mean()))

        # now train...
        for i in range(10000):
            f.train(x,y)

        print('MSE at iteration 10000 is {}'.format(((f.compute(x) - y)**2).mean()))
```

```
MSE at iteration 0 is 10.355688135436504
MSE at iteration 10000 is 0.5883747123982974
```

Now I know I have a working regression network and a working Q-learning algorithm...

Augment the Agent

Now the agent

- is initialized with a replay buffer and a Q network
- has a method to gather experience (build up the replay buffer)
- behaves according to the usual ϵ-greedy policy (note the network call!)

```
class Agent:

    def __init__(self, tf_session):
        self.n_in = 4
        self.n_out = 2
        # first what reward has the agent accrued so far
        self.total_reward = 0
        # discount, learning, exploration rates, batch size
        self.gamma = 0.99
        self.epsilon = 1.0
        self.batch_size = 50
        # make an experience replay buffer
        self.replay_buffer = Replay()
        # make the network that will be the q function
        self.q = Network(tf_session, self.n_in , self.n_out)

    def gather_experience(self, last_observation, action, reward, observation):
        # push this experience onto the replay buffer
        self.replay_buffer.write((last_observation, action, reward, observation))

    def choose_action(self, observation):
        # behave according to an epsilon greedy policy
        if np.random.rand() > self.epsilon:
            if self.q.compute(observation)[0,0]>self.q.compute(observation)[0,1]:
                return 0
            else:
                return 1
        else:
            # explore
            return int(np.round(np.random.random()))
```

Now I know I have a working regression network and a working Q-learning algorithm...

Q-LEARNING

Conceptually (almost) identical

- The same fundamental loop of state, action, reward, state, (q update),...
- Small change: write experience to Agent buffer for later replay
- Small change: write a `None` state to recognize failure (why does replay necessitate this?)
- And a bit of the usual `tf` overhead (without `tb` for clarity)

```
with tf.Graph().as_default():
    ep_rewards = []
    with tf.Session() as sess:
        # create an agent
        agent = Agent(sess)
        # usual tf initialization
        sess.run(tf.global_variables_initializer())
        ####
        # Q-learn (train) DQN on CartPole
        ####
        for ep in range(1501):
            # reset environment and agent
            last_observation = env.reset()
            agent.set_total_reward(0)
            # done at T==199 so no reason to go further
            for t in range(201):
                # agent chooses an action
                action = agent.choose_action(last_observation)
                # agent takes the action, and the environment responds
                observation, reward, done, info = env.step(action)
                # check for fail state
                if done==True:
                    observation = None
                # update agent with reward and data
                agent.gather_reward(reward)
                agent.gather_experience(last_observation, action, reward, observation)
                # update q function, which will use the memory
                agent.q_update()
                # iterate
                last_observation = observation
                if done==True:
                    ep_rewards.append(agent.get_total_reward())
                    break
```

Note the conceptual importance of thoughtful design/abstractions to simplify implementation

UPDATING $Q_\theta(s,a)$

The only novel complexity here is taking steps in θ to optimize $Q_\theta(s,a)$. Recall:

$$\min_\theta \quad (y_i - Q_\theta(s_i,a_i))^2$$

$$\text{where} \quad y_i = \begin{cases} r_i & \text{if } s_{t+1} \text{ is terminal} \\ r_i + \gamma \max_a Q_{\theta^{old}}(s_{i+1},a) & \text{else} \end{cases}$$

In the `Agent` class:

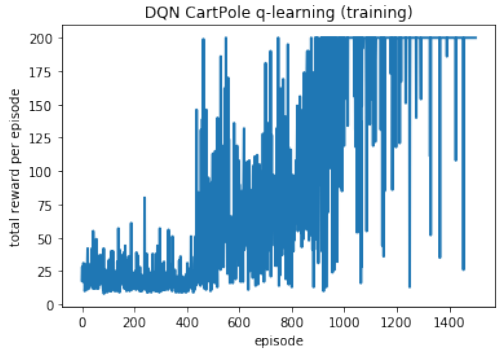
```
32      def q_update(self):
33          # pull a batch from the buffer
34          sars_batch = self.replay_buffer.read(self.batch_size)
35          # compute the q function for all last_obs and obs
36          q_last = self.q.compute([s[0] for s in sars_batch])
37          # q_next for current obs requires a bit more attention, since done flag means q should be zero
38          q_this = np.zeros_like(q_last) # initialize q to zeros
39          ind_not_none = [i for i in range(np.shape(sars_batch)[0]) if sars_batch[i][3] is not None]
40          q_this_not_none = self.q.compute([sb[3] for sb in sars_batch if sb[3] is not None])
41          # now fill q_this with just the valid q, leaving others [0,0]
42          for i in range(len(ind_not_none)):
43              q_this[ind_not_none[i],:] = q_this_not_none[i,:]
44          # a list comprehension is nice but 5x inefficient... want to pass tensorflow a batch block
45          # q_this = [([0,0] if s[3] is None else self.q.compute(s[3])) for s in sars_batch]
46          # now chunk this up as the train_step expects
47          x_batch = np.zeros([np.shape(sars_batch)[0],self.n_in])
48          y_batch = np.zeros([np.shape(sars_batch)[0],self.n_out])
49          for i in range(np.shape(sars_batch)[0]):
50              x_batch[i,:] = sars_batch[i][0]
51              for j in range(2):
52                  if j == sars_batch[i][1]:
53                      # the key step... this is the q learning target
54                      y_batch[i,j] = sars_batch[i][2] + self.gamma*np.max(q_this[i])
55                  else:
56                      y_batch[i,j] = q_last[i][j]
57          # now run the train step
58          self.q.train(x_batch,y_batch)
```

Note:

- exploits/requires the `None` terminal state
- computational efficiency: here sacrificed code clarity for speed ($5-10\times$)
- all `tf` is hidden in `q.train`

LEARNING AND CONTROLLING CARTPOLE WITH DQN

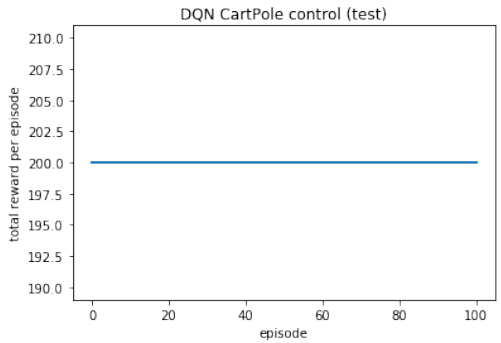
We used an ϵ-greedy behavior policy to *explore* (note: large ϵ found empirically useful in DQN)



Once we have learned $Q_\theta(s, a)$, we now only want to *exploit*. Control without learning:

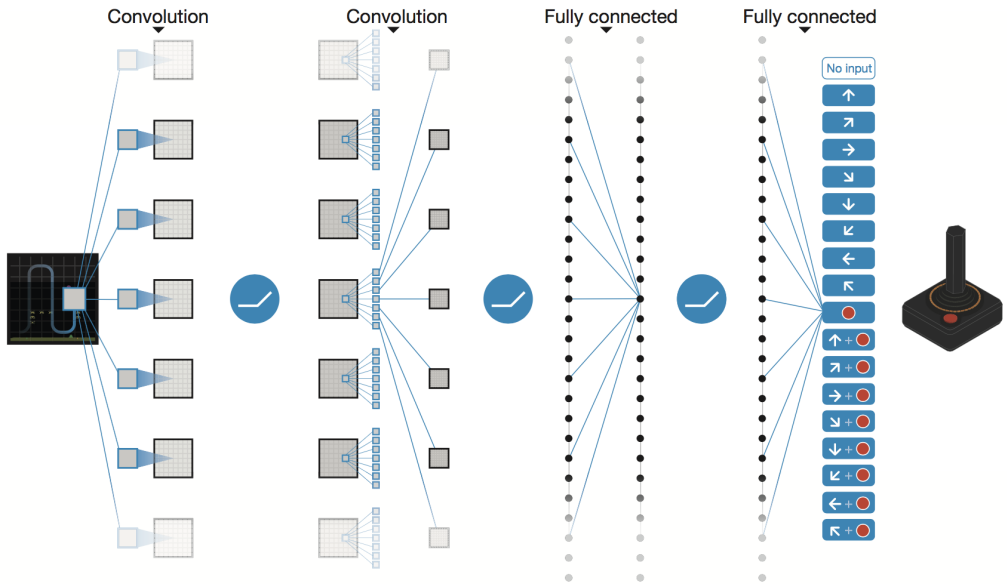
```
50          for ep in range(101):
51              # reset environment and agent
52              last_observation = env.reset()
53              agent.set_total_reward(0)
54              agent.reset_epsilon()
55              # done at T==199 so no reason to go further
56              for t in range(201):
57                  env.render()
58                  action = agent.choose_action(last_observation)
59                  observation, reward, done, info = env.step(action)
60                  agent.gather_reward(reward)
61                  last_observation = observation
```

Greedy performance (render and watch the learned policy)

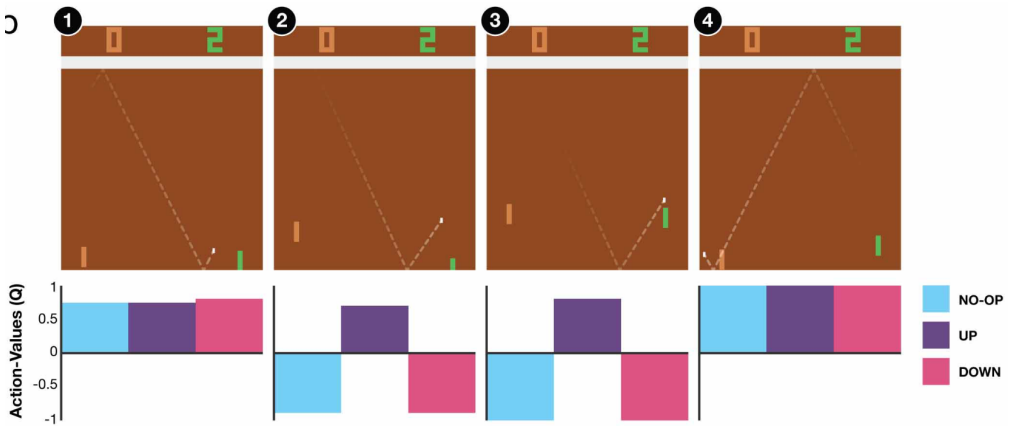


State of the art DQN

From here simply elaborate Q network (includes CNN frontend)



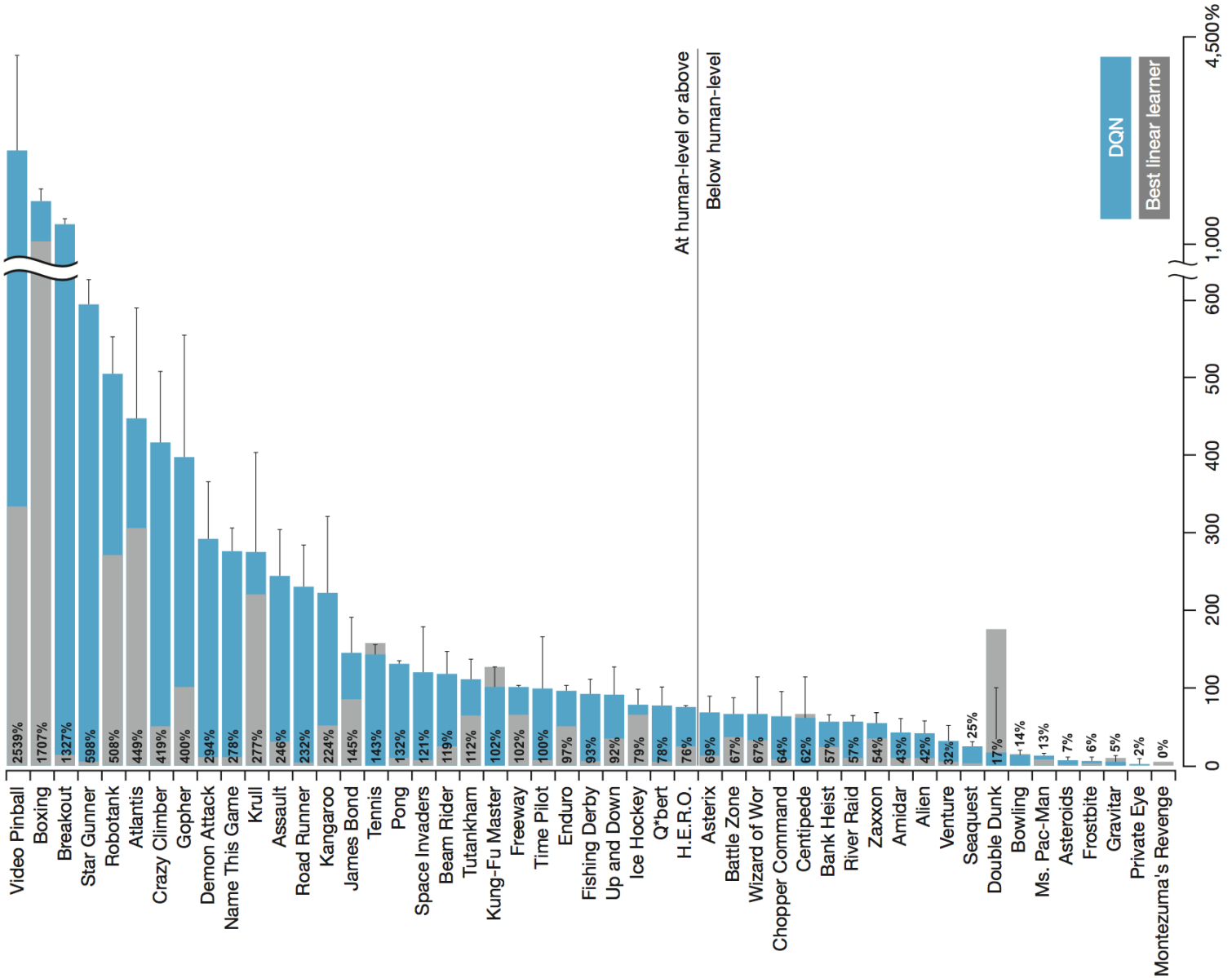
Learn the Q function for Pong



[Mnih et al (2015)]

State of the art DQN

Superhuman performance across a range of different games



[Mnih et al (2015)]

REINFORCEMENT LEARNING: WHERE NEXT

Of course, there is a great deal of underlying empiricism in DQN and RL generally:

- hyperparameter and network adjustment
- training runs and replay buffers
- data preprocessing
- early training policies (eg in CartPole: do better by learning Q_θ from left-right)
- etc...

Where to go from here:

- Consider RL for final project
- Get the Atari emulator in OpenAI gym (`https://github.com/openai/gym#atari`)
- Play with DQN (see `courseworks> files > TF_agents_intro.ipynb`)

  helpful reading: `https://medium.com/analytics-vidhya/tf-agents-a-flexible-reinforcement-learning-library-for-tensorflow-5f125420f64b`
- Proceed to actor-critic methods (`https://github.com/tensorflow/agents`)

RECURRENT NEURAL NETWORKS

TRANSITION TO RNN: RECALL TEXT DATA

Can we predict the next word in a text?

- In language, the co-occurrence and order of words is highly informative.
- This information is called the **context** of a word.
- We can use such a model to generate text of arbitrary length

Example: The English language has over 200,000 words.

- If we choose any word at random, there are over 200,000 possibilities.
- If we want to choose the next word in

  There is an airplane in the __

  the number of possibilities is *much* smaller.

Context information is well-suited for machine learning:

- By parsing lots of text, we can record which words occur together and which do not.
- Reminder (from previous class): the vanilla models based on this idea are *n-gram models*.

BIGRAM MODELS

Bigram model:

- A bigram model represents the conditional distribution

$$\Pr(\text{word}|\text{previous word}) =: \Pr(h_l|h_{l-1}) \ ,$$

- w_l is the lth word in a text.
- Bigram models are a simple Markov chain on words: a *family* of d multinomials, one for each possible previous word.

N-gram models

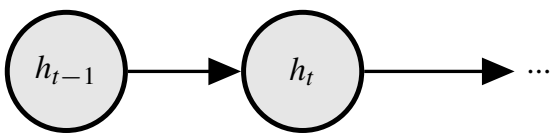
- More generally, a model conditional on the $(N-1)$ previous words

$$\Pr(h_l|h_{l-1}, \ldots, h_{l-(N-1)})$$

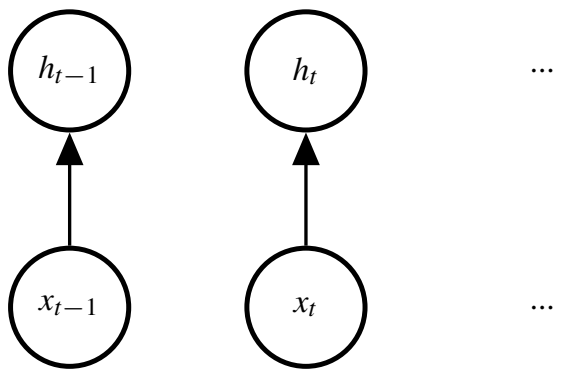
  is called an N**-gram model** (with the predicted word, there are N words in total).
- Unigram model: the special case $N = 1$ (no context information)

Transitioning representations (example bigram model)

probabilistic modelling view



RNN functional view (x_t = prev word)



LEARNING SHAKESPEARE (1)

Unigram Model

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have

Every enter now severally so, let

Hill he late speaks; or! a more to leg less first you enter

Are where exeunt and sighs have rise excellency took of.. Sleep knave we. near; vile like

Bigram Model

What means, sir. I confess she? then all sorts, he is trim, captain.

Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.

What we, hath got so she that I rest and sent to scold and nature bankrupt, nor the first gentleman?

Enter Menenius, if it so many good direction found'st thou art a strong upon command of fear not a liberal largess given away, Falstaff! Exeunt

[Jurafsky and Martin, "Speech and Language Processing", 2009]

LEARNING SHAKESPEARE (2)

Trigram Model

Sweet prince, Falstaff shall die. Harry of Monmouth's grave.

This shall forbid it should be branded, if renown made it empty.

Indeed the duke; and had a very good friend.

Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.

Quadrigram Model

King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in;

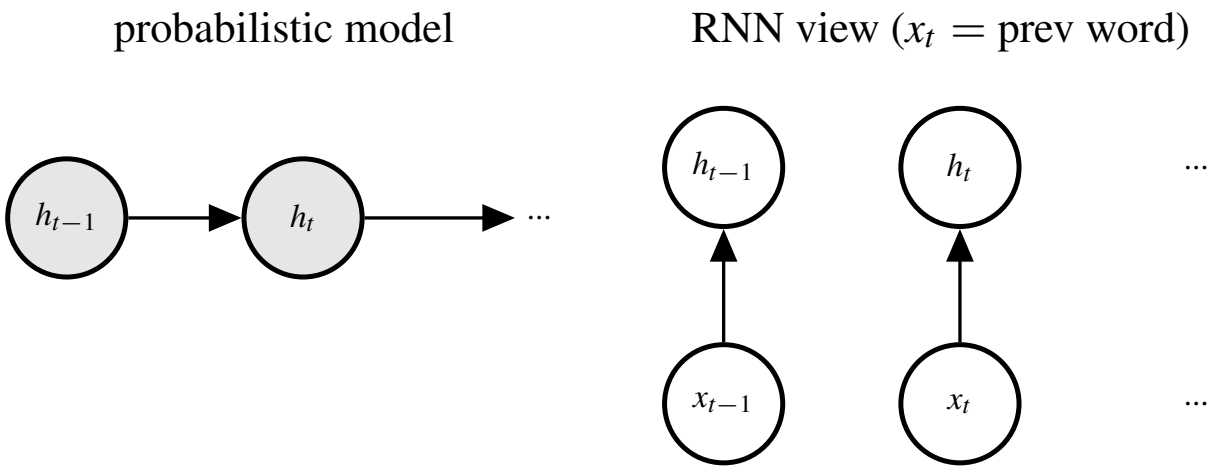
Will you not tell me who I am?

It cannot be but so.

Indeed the short and the long. Marry, 'tis a noble Lepidus.

[Jurafsky and Martin, "Speech and Language Processing", 2009]

Cost



Basic Markov models scale terribly with context size:

- N-gram model considers ordered combinations of N distinct words
- Suppose a text corpus contains 100,000 words. Thus $100000^N = 10^{5N}$ parameters
- As such, N-gram models are conceptually valuable but won't scale
- Long-timescale context is critical. Consider the classic example:

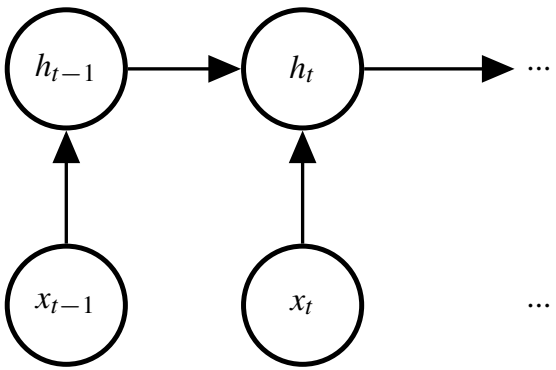
  "I am from California and lived in various places for many years. Therefore I speak __."
- This cost only gets worse for *hidden* Markov models with (possible) inputs

RECURRENT NEURAL NETWORKS

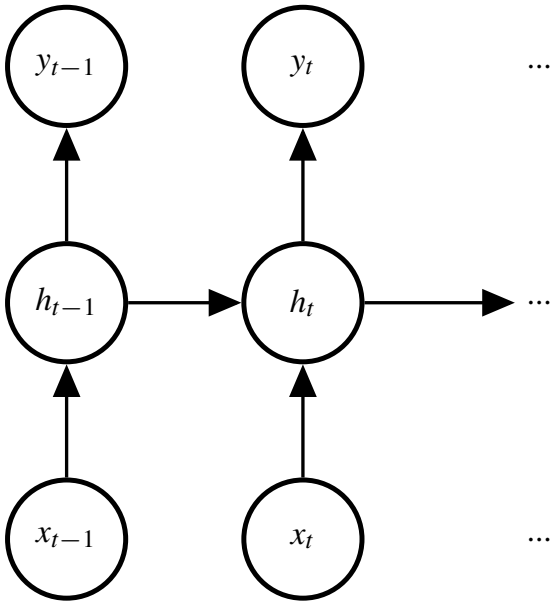
Key idea: $h_t = g_\theta(h_{t-1}, x_t)$. A *hidden state* carries longer-term context information

- RNNs use a neural network for this evolution of hidden state (but it needn't be)
- A *single, fixed* network g_θ governs transitions (cf. HMM transition matrix)

Output can be h_t



Output can be $y_t|h_t$ (cf. Markov model vs HMM)



Warning:

- There is rarely agreement on what a particular structure means (eg LSTMs; cf. CNNs)
- There is no definitive text (though many papers) articulating these concepts
- ...but RNNs are rapidly evolving and producing some of the most exciting results in AI

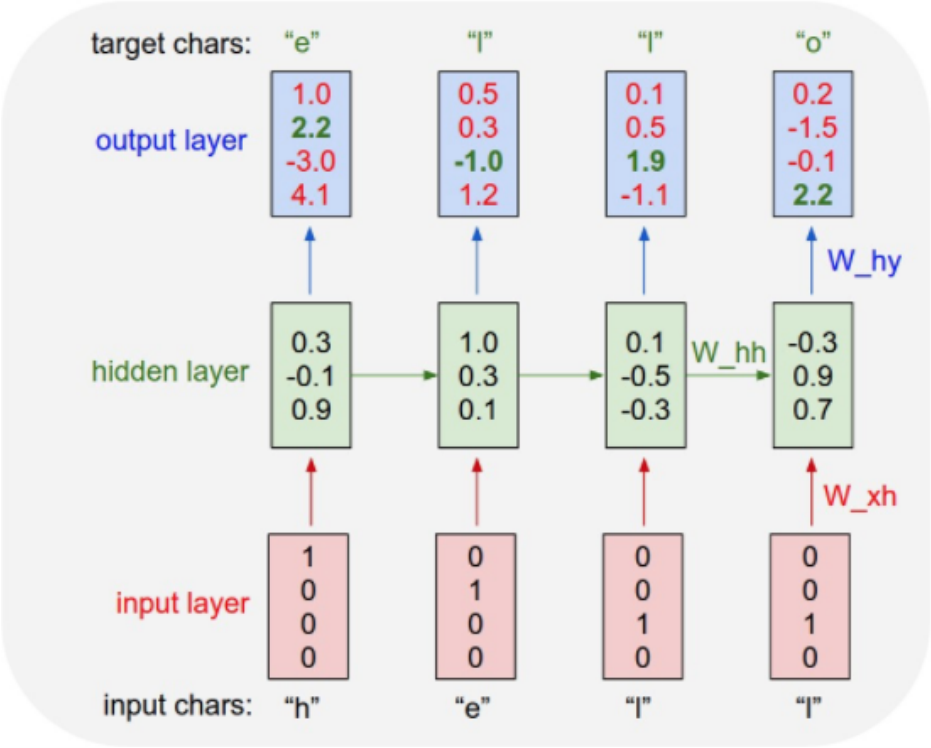
RNN SIMPLE EXAMPLE

Consider the following simple *character* model:

- alphabet consists of $\{h, e, l, o\}$, one-hot encoded
- hidden layers evolve as $h_t = \sigma\left(W_{hh} h_{t-1} + W_{xh} x_t\right)$

... (σ is usual activation nonlinearity, here `tanh`)

- output $y_t = W_{hy} h_t$ (think `logits`... then take softmax)



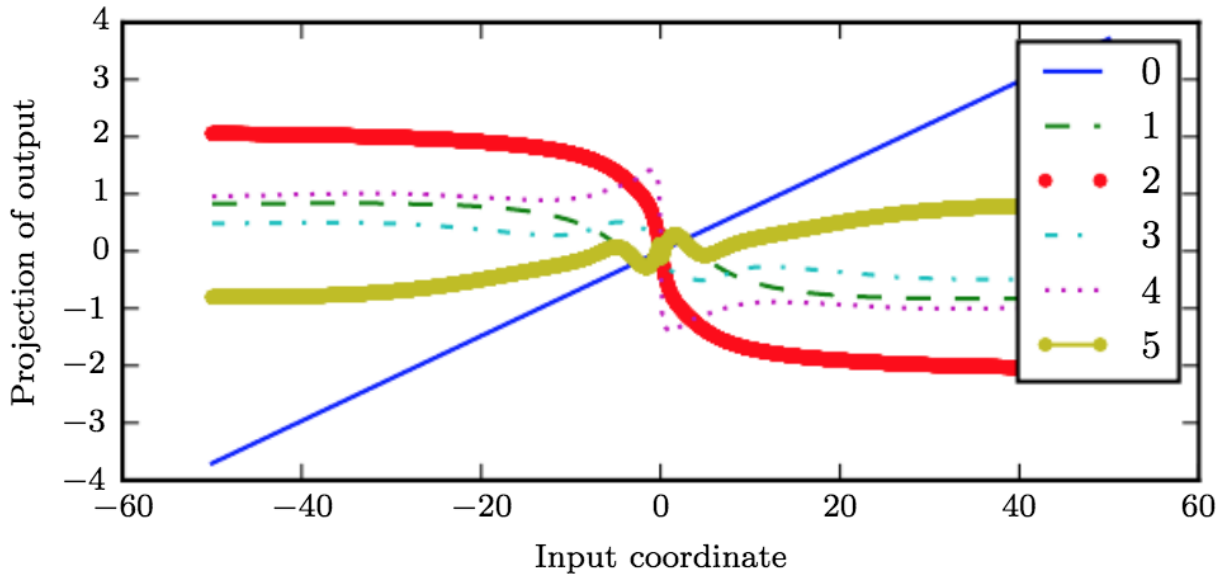
`http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Intent: h_t carries longer-range context, without exponential parameters of N-gram models.

VANISHING GRADIENTS

Recall the vanishing gradient discussion from deep CNNs:

- Backprop is the chain rule, multiplying Jacobians together repeatedly
- Exponential decay of gradients results
- Simple demonstration: repeated linear/tanh/linear/tanh/...



[Goodfellow et al 2016, ch10]

- Particularly relevant in RNNs: long-range context ignored over short-range

Much work has gone into designing clever network structures to persist long-range context

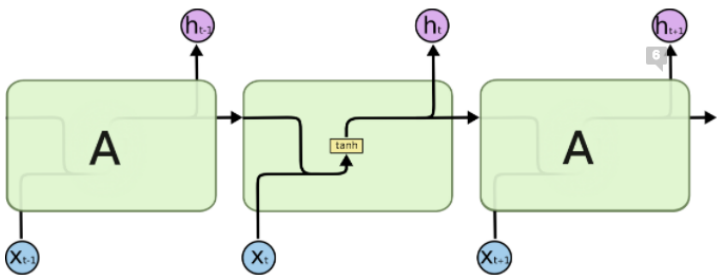
LONG SHORT-TERM MEMORY NETWORKS

Long Short-Term Memory Networks are the de facto standard for RNN memory context

- Custom engineered network architecture to have a notion of memory
- (recall CNNs: hand-chosen architecture to exploit problem structure)
- Origin [Hochreiter and Schmidhuber 1997]; many times improved and iterated since then
- Only recently (2014) has a second major alternative architecture arisen (next class)

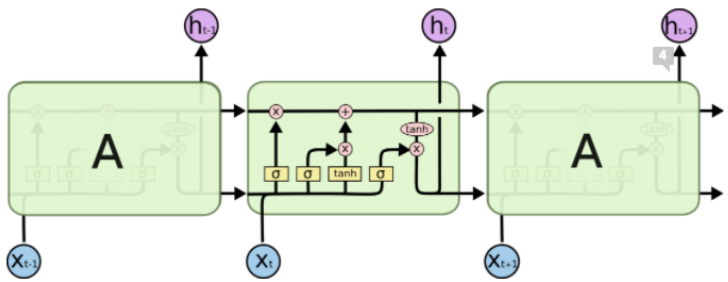
Understand the abstraction: there is simply a network g_θ evolving hidden state

Original RNN



$$h_t = tanh\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

Full LSTM



$$\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}$$

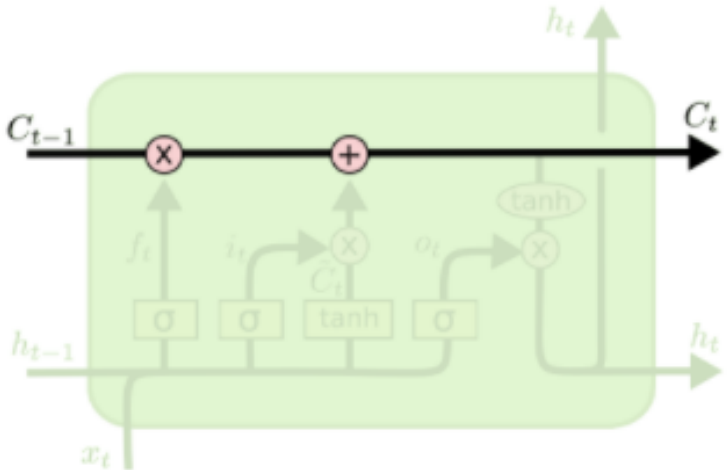
Pictures from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Notation consistent with [Jozefowicz et al 2015]

LSTM CELL STATE

Rather than hidden state h_t, we now pass h_t and a *cell* state c_t

- This is no problem: define $\bar{h}_t \triangleq \begin{bmatrix} h_t \\ c_t \end{bmatrix}$, and it is still an RNN.



The cell state:

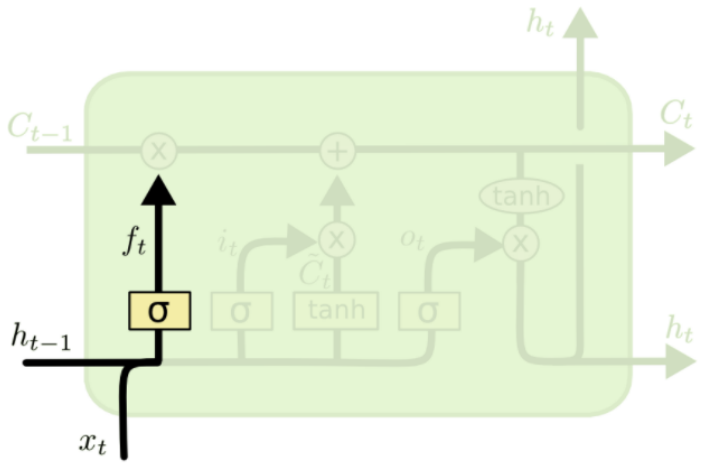
- provides a channel for long-range information/memory to propagate forward
- without corrupting/compromising the hidden state (which is directly output relevant)

Note: the LSTM network architecture is often (inconveniently?) called an LSTM *cell*.

LSTM FORGET GATE

Now we must consider how the hidden state and cell state interact. First, the *forget gate*:

- Conceptually, f_t chooses to forget or pass the current cell state
- Elementwise forgetting, so it is doing so individually for each unit (the width) of c_t



$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right)$$

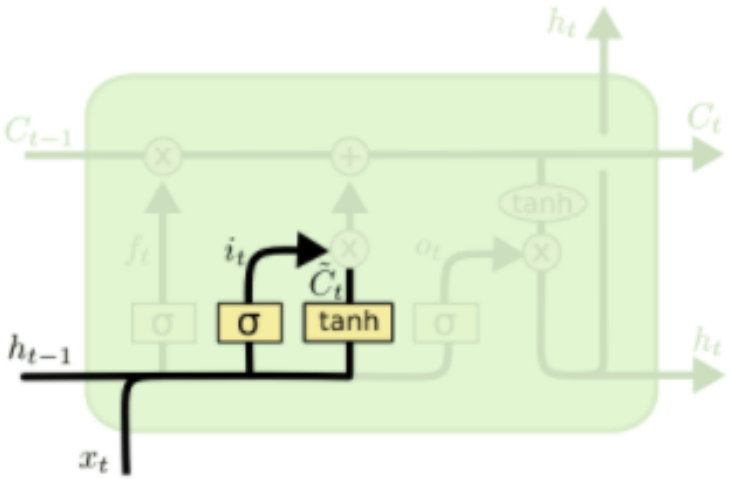
The forget gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that remove or persist certain dimensions of c_t
- Convince yourself that this is a useful way to free or hold data in memory
- Note: σ must be $\in [0, 1]$, but can be sigmoid, tanh, etc...

LSTM INPUT GATE

Continuing hidden state and cell state interaction. The *input gate*:

- If f_t chooses to forget or pass the existing cell state...
- Input i_t chooses what to pass in as a new cell state
- Again elementwise...



$$
\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)
\end{aligned}
$$

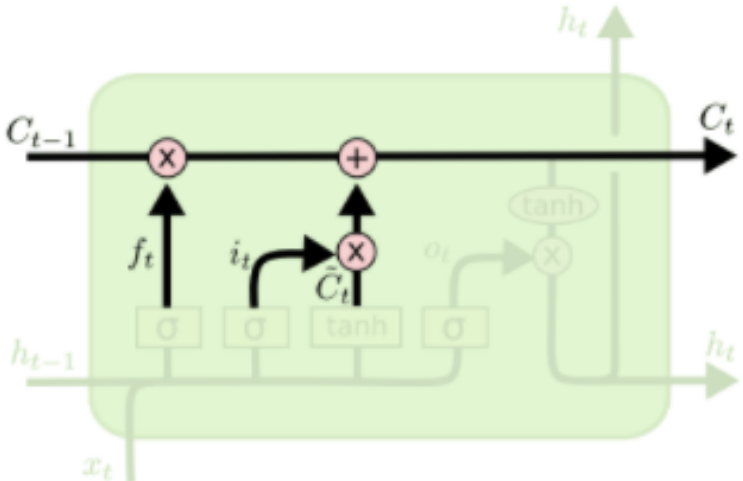
The input gate

- can be thought of as projecting dimensions of x_t and h_{t-1}
- ... that load or ignore certain dimensions of the new proposed cell state $\tilde{c}_t$
- Convince yourself that this is a useful way to load/not load data into memory
- Note: again σ must be $\in [0, 1]$, but can be sigmoid, tanh, etc...

LSTM CELL STATE AGAIN

The effects of the forget and input gates are then loaded onto the cell state c_t:

- Elementwise action of persisting/overwriting the long-term memory cell c_t



$$c_t = c_{t-1} \odot f_t + \tilde{c}_t \odot i_t$$

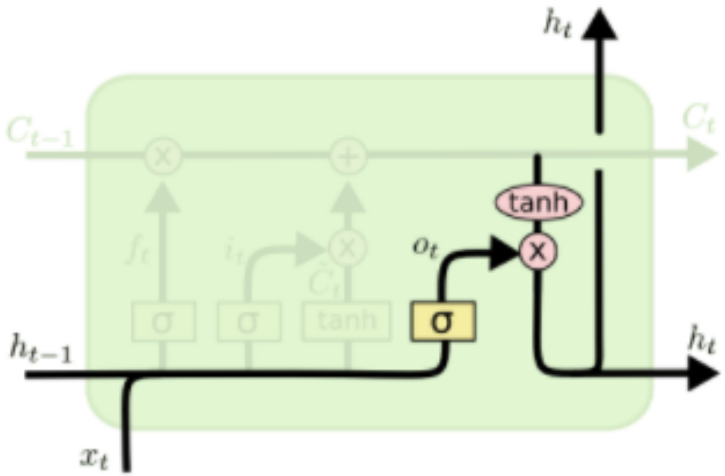
Critical to intuition:

- This is neural networks, so we hope to *learn* from data when to forget, load, etc.
- All operations here are elementwise, so many different loads/persists occur in parallel
- So far we haven't affected h_t yet...

LSTM OUTPUT GATE

Continuing hidden state and cell state interaction, but now to h_t. The *output gate*:

- If f_t chooses to forget or pass, and i_t chooses what to pass...
- o_t chooses when to write out the cell c_t to h_t.



$$
\begin{aligned}
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}
$$

Same as before: the output gate is a useful way to send data onto h_t

Note the key and complementary differences here between h_t and c_t;

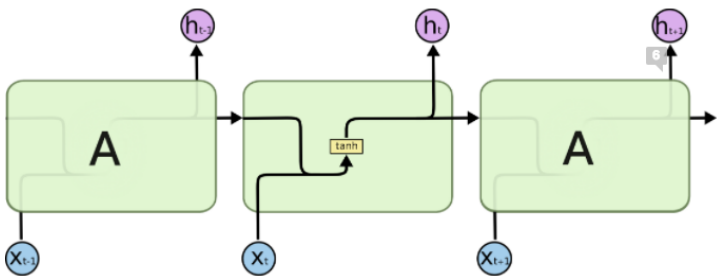
- h_t is either the output or parameterizes the output $y_t|h_t$.
- h_t thus has short-term or more immediately relevant data
- c_t can persist over long-range periods and needn't (directly) drive output (o_t)

LONG SHORT-TERM MEMORY NETWORKS

We have built up the structure of a standard LSTM

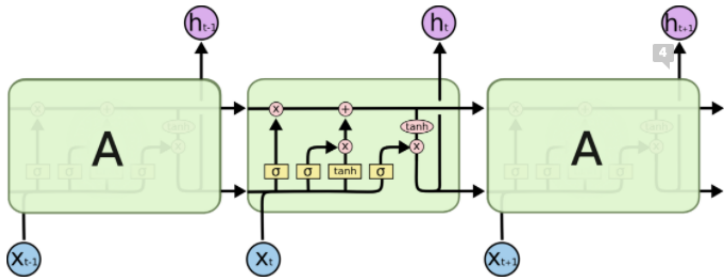
- there are many minor variants
- but all share the basic forget/input/output and cell/hidden components
- thankfully, neural network libraries abstract all these blocks and parameters away
- See for example `tf.contrib.rnn.LSTMCell`
- The key reminder: like a CNN, this is just a (highly engineered) neural network g_θ

Original RNN



$$h_t = tanh\left(W_{xh}x_t + W_{hh}h_{t-1} + b_h\right)$$

Full LSTM



$$\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t &= tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t &= c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t &= tanh\left(c_t\right) \odot o_t
\end{aligned}$$

SHAKESPEARE DATA

We will treat all of Shakespeare as a long string

```
...
COMINIUS:
It is your former promise.

MARCIUS:
Sir, it is;
And I am constant. Titus Lartius, thou
Shalt see me once more strike at Tullus' face.
What, art thou stiff? stand'st out?

TITUS:
No, Caius Marcius;
I'll lean upon one crutch and fight with t'other,
Ere stay behind this business.
...
```

This string:

- has length 4573338
- can be one-hot encoded with vectors $x_i \in \mathbb{R}^{67}$, namely:

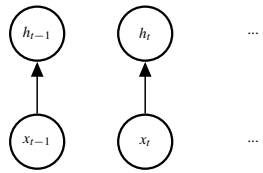
```
The 67 inputs are:
 ['O', '?', 'G', 'k', 'Y', 'W', 'L', 's', 'i', 'T', 'v', '&', '3', ']', 'f', '-', ',', 'c', 'C', 'J', 'x', '!', 'F',
'$', 'D', 'B', 'R', 'b', 'o', 'e', 'S', ':', "'", 'E', 'h', 'V', 'z', 'y', '\n', '.', 'l', 'a', 'j', 'q', 'P', 'U',
'[', 'M', 'A', 'N', 'g', 'd', 'X', 'I', 'w', 'K', ' ', 'H', 'm', 'Q', 't', 'p', ';', 'n', 'u', 'Z', 'r']
```

Recall N-gram models on words. Now we model Shakespeare *character by character*

RNN ANALOGY TO A BIGRAM MODEL

Recall:

- Each x_t is the previous character (context!)
- Network predicts h_t from x_t
- No recurrence here (yet)...




```
----Post-training Sample----
 pawhenyyrcato he f to avyrod
T: couwendory:

s wEI :
Tt
ILouthe hair'le,e er s the;Kt t t u
```

Notice:

- This is multinomial, so we can sample characters from the network output
- Try an easier dataset:

```
----Pre-training Sample----
 nodz nppvqfvfu qfyxbrvmathpengrlvgkqtlaozzdct otfrwdekrkdp wircabmcaxwntgvnkwlvqgxyaweuawxm
----Post-training Sample----
 ick juick fog oved fog the jumped jumpe rown jumpn quick brog the  jumpe therown fove fown
```

- We could also predict with a more straightford `np.argmax`

```
----Pre-training Sample----
 rfvydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydydyd
----Post-training Sample----
 jumpe the the the the the the the the the the the the the the the the the the the the the the the t
```

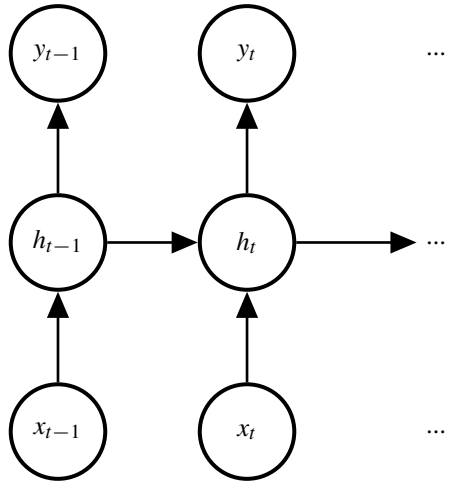
BACKPROPAGATION THROUGH TIME

As usual we seek to take gradients in θ:

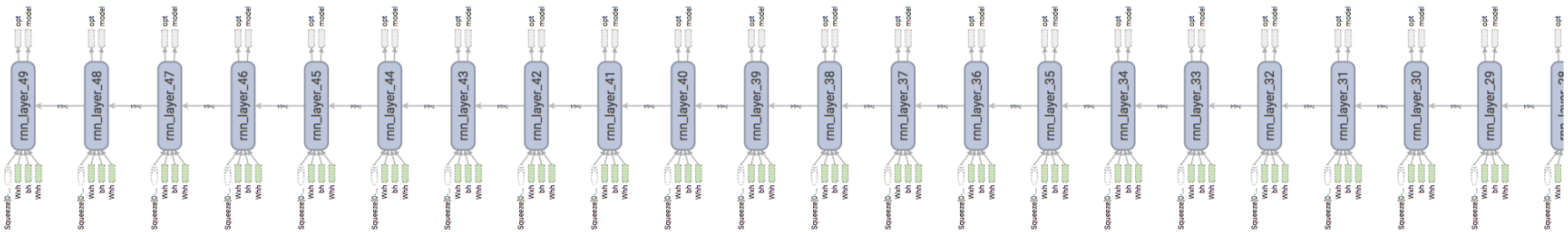
```
# define RNN
self.Wyh = tf.get_variable('Why', shape=[self.n_hidden,self.n_out])
self.by = tf.get_variable('by', shape=[self.n_out])
if self.rnn_type=='1layer':
    self.Wxh = tf.get_variable('Wxh', shape=[self.n_in,self.n_hidden])
    self.bh = tf.get_variable('bh', shape=[self.n_hidden])
    self.Whh = tf.get_variable('Whh', shape=[self.n_hidden,self.n_hidden])
```

```
def rnn_layer(self,x,h):
    with tf.name_scope('rnn_layer'):
        # this can be called either via training or stepping
        if self.rnn_type=='1layer':
            return tf.nn.tanh(tf.matmul(x, self.Wxh) + tf.matmul(h, self.Whh) + self.bh)
```

```
def rnn_logit(self,h):
    # called either via training or stepping
    with tf.name_scope('rnn_logit'):
        return tf.matmul(h, self.Wyh) + self.by
```



But wait...



Context:

- Though $|\theta|$ is manageable, the chain rule can extend arbitrarily far back in time
- We will truncate at some length (here $T = 50$) and call that the *context* of h_t
- We believe that this depth will provide adequate approximation to the true gradient...

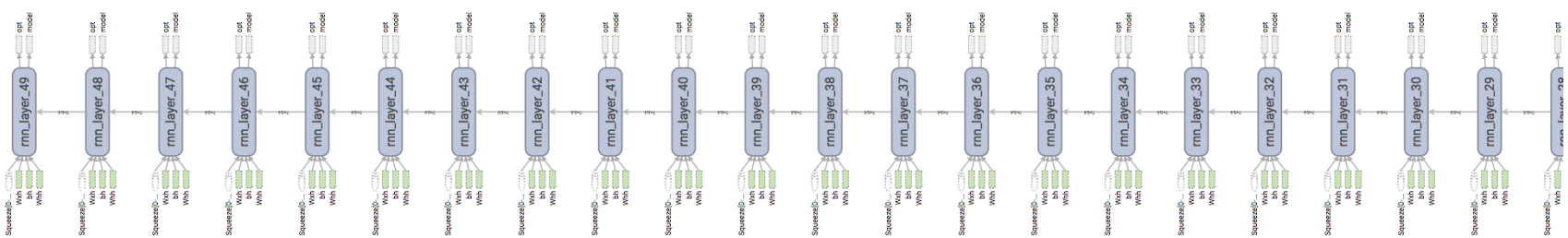
CONTEXT IN `tensorflow`

We will train on *context batches* of length $T = 50$ (or similar)

- Unlike all previous batching, context batches are sequential
- `tf` must loop through to propagate the hidden state h_t

```
# split (and squeeze) to get BPTT inputs, that is, a list of length n_context with usual [batch_size,n_in]
# note: see code at bottom of notebook for critical ",[1]" fix
self.xs = [tf.squeeze(xx,[1]) for xx in tf.split(self.x, self.n_context, axis=1)]
self.ys = [tf.squeeze(yy,[1]) for yy in tf.split(self.y, self.n_context, axis=1)]

# propagate h through context length
self.h = []
h = self.h_
for x in self.xs:
    # here the first time h_ is broadcast to the np.shape(x,0) (as in, batch_size)
    h = self.rnn_layer(x,h) #tf.nn.tanh(tf.matmul(x, self.Wxh) + tf.matmul(h, self.Whh) + self.bh)
    self.h.append(h)

# make outputs from h
with tf.name_scope('model'):
    self.logits = []
    self.ypred = []
    for h in self.h:
        logits = self.rnn_logit(h)
        self.logits.append(logits)
        self.ypred.append(tf.nn.softmax(logits))
```

`self.rnn_layer` carries *same* parameters, but h_t is now recurrent and can now be trained:



1 LAYER RNN TRAINED ON SHAKESPEARE

Notes:

- Iterations are each batches of $T = 50$ context, sequentially, with $h_0 = [0, \ldots, 0]$
- Effectively 7 *epochs* (full passes through text)
- Single hidden layer with $n = 64$ units, fully connected to logits (here $\in \mathbb{R}^{67}$)
- Accuracy/loss is averaged over batch in the usual way
- Learning occurs, and frankly high accuracy is unlikely (even undesirable?)

FORWARD SAMPLING TEXT

Consideration:

- How to forward sample text?
- Where do we get h_{t-1}?
- How to step $+1$ when we wrote the code to operate on a context of depth $T = 50$?

```
k = (epoch*batches_per_epoch + batch).astype(int)
summary_writer.add_summary(summary, k)
print('______[epoch:{},batch:{},all batches:{}] has loss {}______'.format(epoch,batch,k,loss))
# take the last hidden and target to seed a writing
h = h_prev
text_out = y_batch[-1]
for j in range(200):
    # roll forward and fantasize text of length 200
    h, y = rnn.sample_step(text_out[-1],h, sample=True, temp=min(batch/5000,5))
    text_out += y
print(text_out)
print('')
```

Now the RNN can fantasize Shakespeare texts...

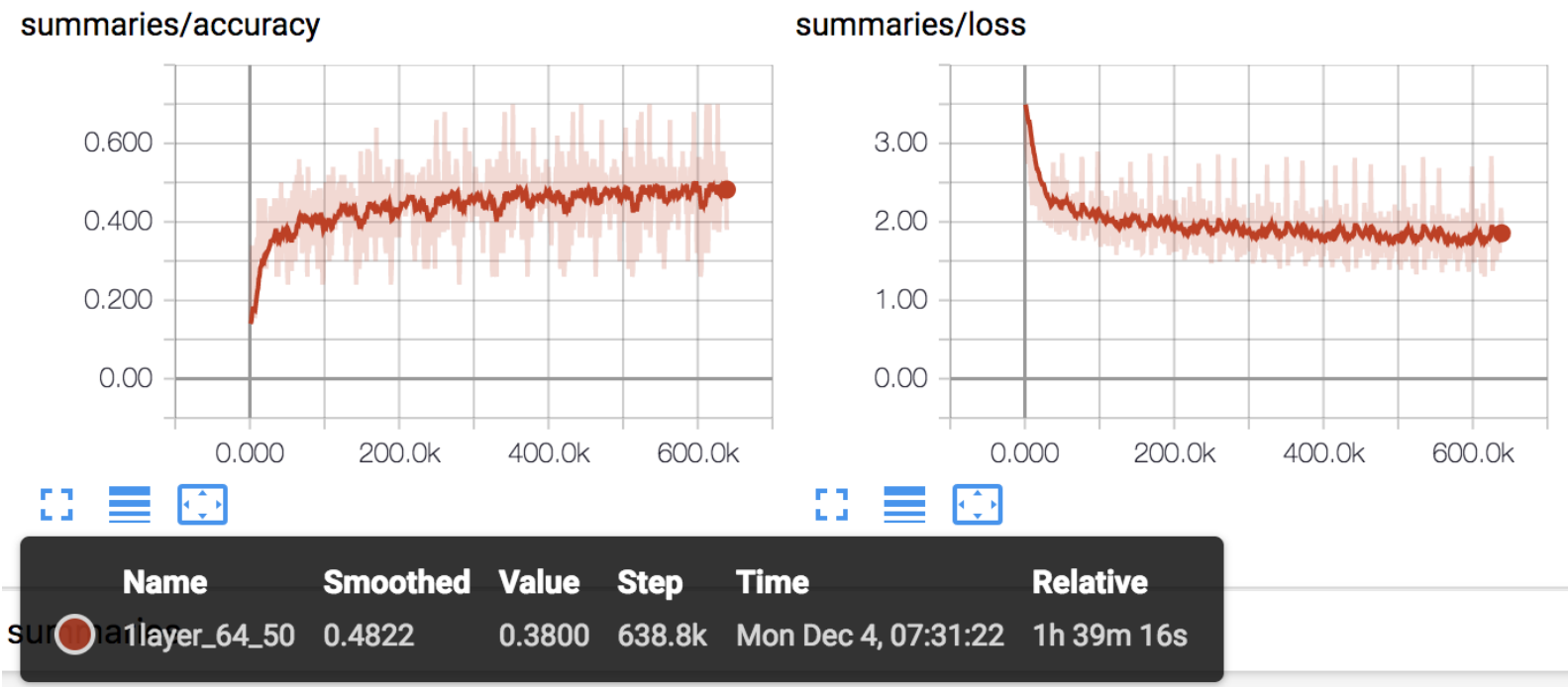
1 LAYER RNN TRAINED ON SHAKESPEARE

Very early in training:

```
_____[epoch:0,batch:6000,all batches:6000] has loss 3.277571439743042_____
 do si, pur et hirb ond aopm bohcon mttt ahr home we, peme thaucno, ior rere lethe mias iol lh

wtye thot Toates ases n wnmdsd tott anl mhew shers thie caeuame soece cUpfng-r Sowsedt mo tiree
m oie the
```

Later in training:

```
_____[epoch:3,batch:21000,all batches:295398] has loss 1.7853922843933105_____
And sin, I will and have my love the seet the singed the sear and the wart,
The still the have you the singly and that his a dider his and and the have to her for the still and the mangers
And the hav
```



USING THE RNNCell ABSTRACTION IN tf

Tensorflow has an excellent abstraction to handle all the recursion... if you know how to use it.

```
self.c_ = tf.placeholder(tf.float32, [None,self.n_hidden], name='c_')
self.h_ = tf.placeholder(tf.float32, [None,self.n_hidden], name='h_')
# An LSTMStateTuple that can be fed as initial_state to dynamic_rnn
self.state_ = tf.nn.rnn_cell.LSTMStateTuple(self.c_, self.h_)  # 2 x None x n_hidden

# define RNN
self.Wyh = tf.get_variable('Why', shape=[self.n_hidden,self.n_out])
self.by = tf.get_variable('by', shape=[self.n_out])
self.cell = tf.contrib.rnn.LSTMCell(self.n_hidden)
# If cells are LSTMCells state will be a tuple containing a LSTMStateTuple for each cell.
h_outs, self.state_out = tf.nn.dynamic_rnn(self.cell, self.x, initial_state=self.state_)
# time_major=True implies time, batch, depth; see https://www.tensorflow.org/api_docs/pytho
# time_major=False implies batch, time, depth

# now h_outs is batch,time, hidden size
self.h = tf.reshape(h_outs,[-1,self.n_hidden])
with tf.name_scope('model'):
    self.logits = self.rnn_logit(self.h)
    self.ypred = tf.nn.softmax(self.logits)
```

Be careful with `LSTMStateTuple`; know why and how to use it

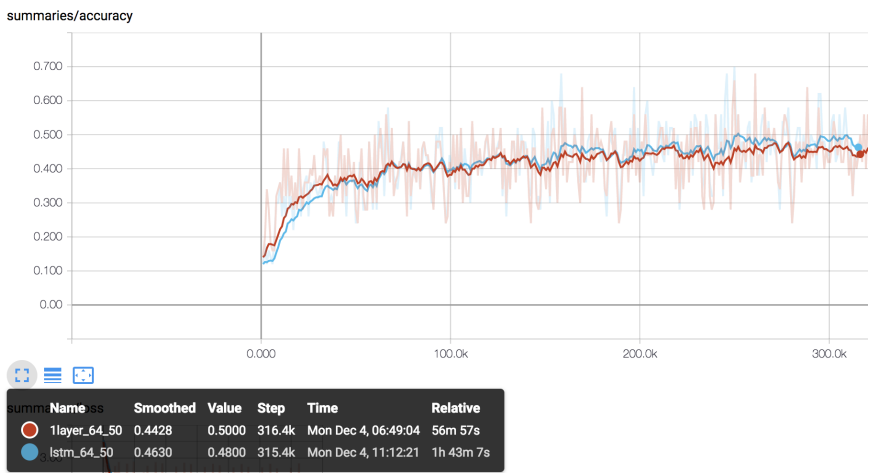
SIMPLE LSTM TRAINED ON SHAKESPEARE

Very early in training:

```
_______[epoch:0,batch:6000,all batches:6000] has loss 3.478269338607788_______
 vh ho osnth twh  eain r ovs shutn haoe   hyr lh he oonctlerk

 aa sEddh serotste
nue ls ldlhe uI hee ds voosit eanuu e sttsht ohme t e’nhcd trost
ti tewe le?,o hus:ee pero rh so  heetbtuy m oteimnowny
```

Later in training:

```
_______[epoch:3,batch:21000,all batches:295398] has loss 1.5456037521362305_______
And the stanter to the well the stange.

PRINCE:
I wall me the with a marter to the sir.

PRINCE:
He sould the with a tould and the sould here
The lear and the words and the sell the werts.

PRINCE:
An
```

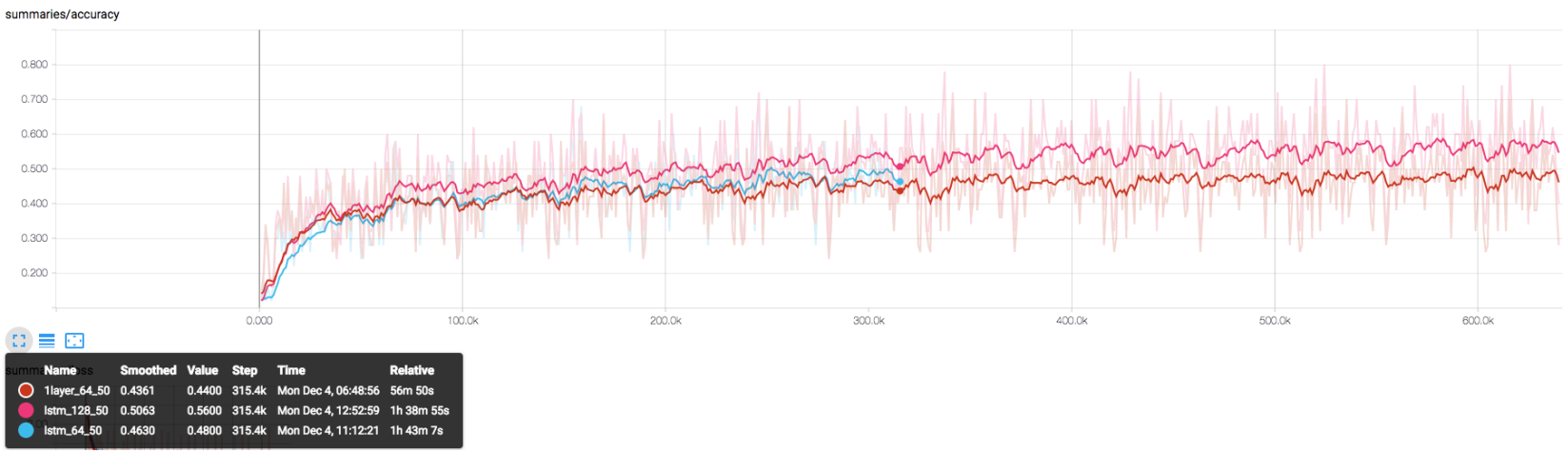


BETTER LSTM TRAINED ON SHAKESPEARE

Trained on character sequences alone!

```
_______[epoch:6,batch:80000,all batches:628796] has loss 1.6592674255371094_______
uch a stranger to see thee and the word.

APEMANTUS:
And there is not for the tooth that we may be so
must be a more and the man and man the soor
And the field to my lord of the company.

TIMON:
The so
```

```
_______[epoch:6,batch:83000,all batches:631796] has loss 1.1526007652282715_______
John, the world
That will be seen the sense of the world,
And the shall be the stranger than the hand
That we shall be a brother to be the word.

PISANIO:
I will not the father than the strong of his g
```

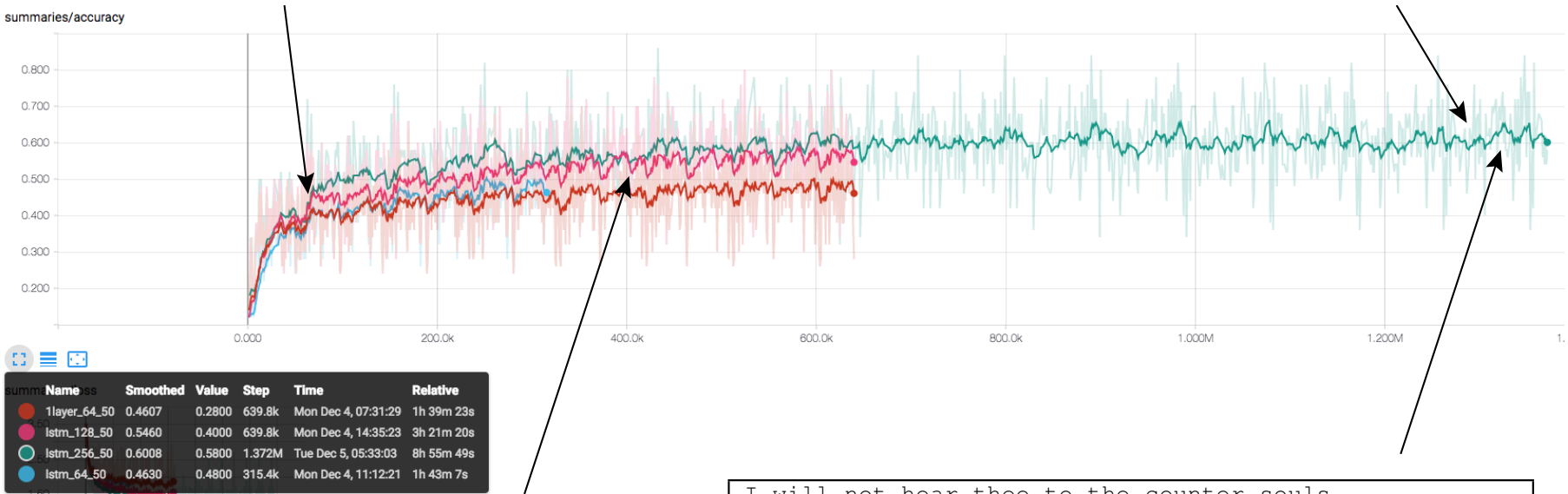


BIGGER LSTM, TRAINED LONGER

256 unit LSTM trained for 15 epochs

```
l the the the cound the serest the here.

CARONES:
The will and the the the come the gorters and
And the hare the there the shere the pranged
The lave the manter the the could with the shere
And the co
```

```
QUEEN MARGARET:
I will not be a man that have been clothes
And have the false than the fortunes of them.

QUEEN MARGARET:
I will not be a state of men and thee,
And therefore like a curse of the best
```



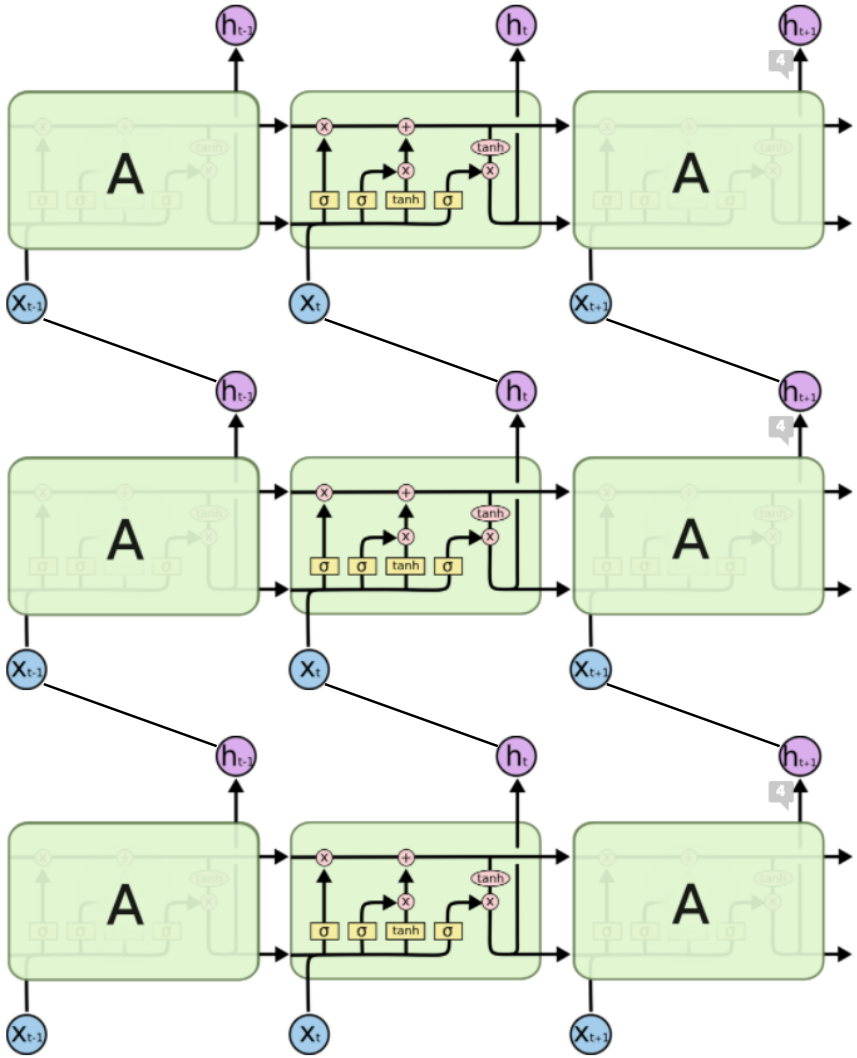

```
 You shall see the state of the charge of the
streather of the moon of the proceased with him.

KING LEAR:
Why, they are not so not the hold him to me,
The preating perceive the good field of the
sense
```

```
 I will not hear thee to the counter souls.

Clown:
What is this thing?

SIR TOBY BELCH:
I will not think the streets of my foes and the state
of this and that thou art a good and beard.

SIR TOBY BELC
```

INCREASING EXPRESSIVITY WITH STACKED LSTM

How to go further:

- LSTM are an input-output function...
- ...so can be composed...
- Elaborate to *stacked* LSTM cells.



Tensorflow makes this easy:

```
cell = tf.contrib.rnn.LSTMCell(n_hidden)
stack = tf.nn.rnn_cell.MultiRNNCell([cell]*n_layers)
```

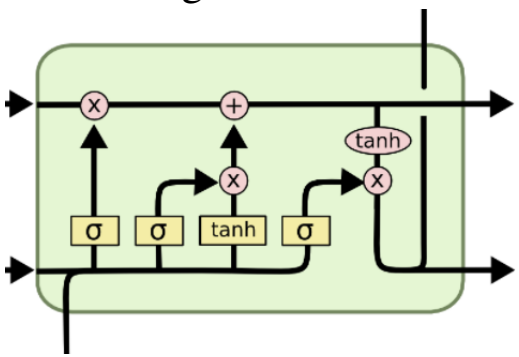
Stacked LSTM and their variants are the workhorse of modern AI with sequence data.

GATED RECURRENT UNITS

Notice

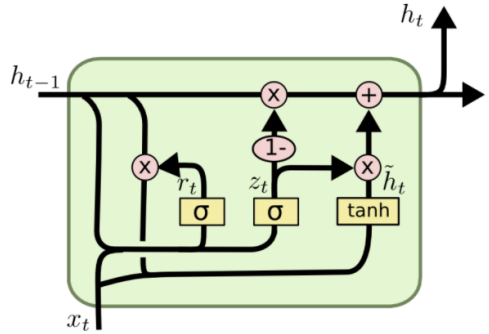
- LSTM offers major increases in performance and long-range dependency modeling
- That said, it's bit difficult to argue the necessity of f_t, i_t, o_t in the LSTM
- Other choices, based on update gate z_t, form the Gated Recurrent Unit [Cho et al 2014]

Original LSTM



$$
\begin{array}{rcl}
f_t & = & \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \\
i_t & = & \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) \\
\tilde{c}_t & = & tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t & = & \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) \\
c_t & = & c_{t-1} \odot f_t + \tilde{c}_t \odot i_t \\
h_t & = & tanh\left(c_t\right) \odot o_t
\end{array}
$$

Gated Recurrent Unit (GRU)



$$
\begin{array}{rcl}
z_t & = & \sigma\left(W_{xz}x_t + W_{hz}h_{t-1} + b_z\right) \\
r_t & = & \sigma\left(W_{xr}x_t + W_{hr}h_{t-1} + b_r\right) \\
\tilde{h}_t & = & tanh\left(W_{xh}x_t + W_{hh}\left(r_t \odot h_{t-1}\right) + b_h\right) \\
h_t & = & (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{array}
$$

Does this matter/help? An ongoing debate:

- See [Jozefowicz et al 2015] for a thorough empirical comparison of architectures
- There is no theory to suggest these choices, though sensible, are necessary or precise
- Try it yourself: compare `tf.contrib.rnn.LSTMCell` to `tf.contrib.rnn.GRUCell`

RECURRENT NEURAL NETWORKS: WHERE NEXT

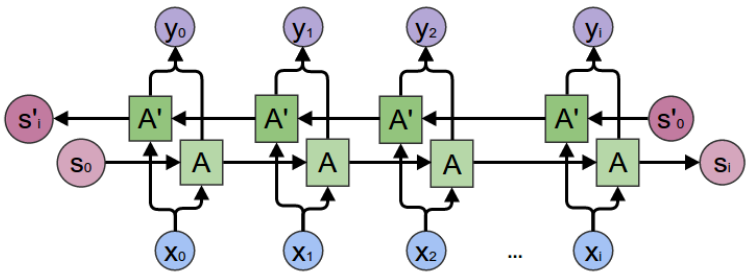
Many of the usual tricks are essential to RNN performance

- validation data, batch normalization, dropout, etc...

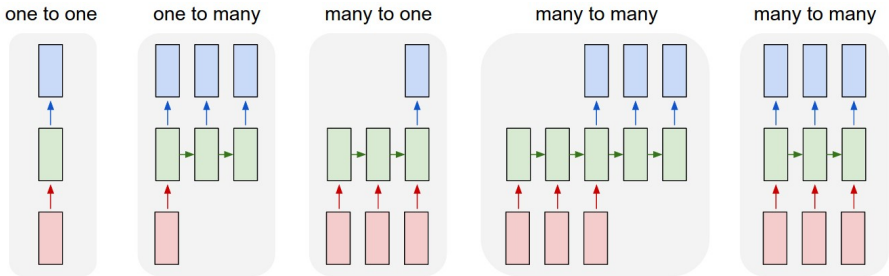
...conveniently: `tf.nn.rnn_cell.DropoutWrapper(cell, output_keep_prob=0.8)`

Where to go next / key ideas that we have not covered:

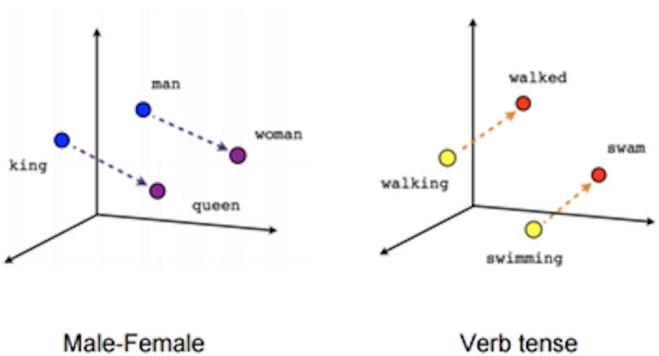
Bidirectional RNNs



Multi-input/multi-output (e.g. seq2seq)



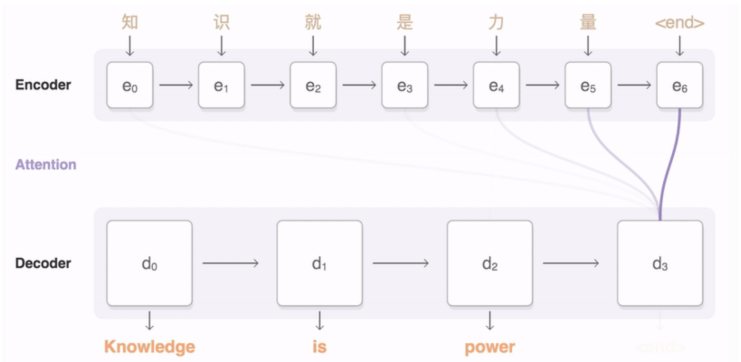
Word embeddings (e.g. word2vec)



Male-Female

Verb tense

Attention



RNNs are a massive area of current and exciting development

Implicit Probabilistic Models a.k.a. Deep Generative Models

MODELING

A central problem in statistics and machine learning is choosing a *model*:

$$\mathcal{M} = \{p_\phi : \phi \in \Phi\}$$

Prescribed probabilistic models:

- form $p_\phi(x)$ directly
- Most of statistics (and what we've seen in these courses) is of this form
- Gaussian, uniform, ...

Implicit probabilistic models:

- Partition the randomness and the structure into two different problems
- Generate *latent* $z_i \sim p_0(z)$ and compute $x_i = g_\phi(z_i)$ with some parameterized function g_ϕ
- Induces a (possibly) more complex model/family of distributions $p_\phi(x)$
- You have seen this before in your first stats class (inversion sampling):

$$z \sim \mathit{Unif}(0,1) \quad x = F_\phi^{-1}(z) \quad \rightarrow \quad x \sim \mathit{Exp}(\phi)$$

F_ϕ is the cdf of the exponential distribution, $F_\phi(x) = 1 - \exp(-\phi x)$, with $F_\phi^{-1}(z) = -\phi \log(1-z)$

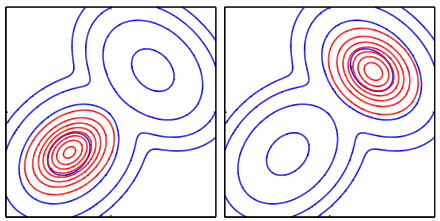
- Natural setting in differential equations, ecology, weather, finance, and many other fields

IPMs WITH DEEP NEURAL NETWORKS

Idea

- Sample randomness from a particularly easy distribution $z \sim \mathcal{N}(0, I)$
- Use a deep neural network as the structure map g_ϕ
- Best of both worlds? ...flexible, expressive $p_\phi(x)$ that is easy to sample and learn

1. Variational inference $q_*(z) = \arg\min_{q \in Q} KL(q||p)$



- Today: the *variational autoencoder* of [Kingma and Welling 2014]

2. Generative modeling

$$z_i \sim \mathcal{N}(0, I) \quad \rightarrow \quad g_\phi(z_i) \quad \rightarrow$$

- Today: the *generative adversarial network* of [Goodfellow et al 2015]
- The paper [Mohamed and Laksminarayanan 2016] clarifies particularly well

RECALL VARIATIONAL INFERENCE

We want to solve an inference problem where the correct solution is an "intractable" distribution with density $p(z|x)$ (e.g. a complicated posterior in a Bayesian inference problem):

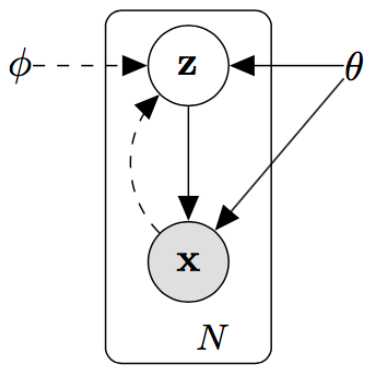
- We stipulate a variational model (a family of simpler distributions) $Q = \{q_\phi(z|x) : \phi \in \Phi\}$
- If the posterior density is $p(z|x) = \frac{p(x|z)p(z)}{p(x)}$, then

$$q^*(z|x) = \arg\min_{q \in Q} KL(q(z|x)||p(z|x)) \ .$$

- Approximate a complicated distribution with the closest member of a tractable family

The ELBO (evidence lower bound) objective:

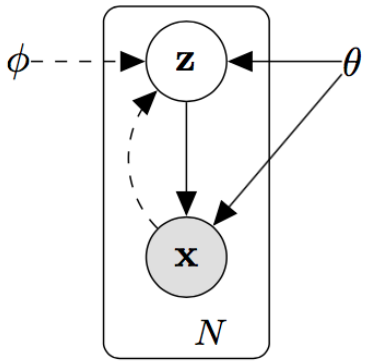
$$\begin{aligned} KL(q(z|x)|p(z|x)) &= E_{q_\phi}\left(\log \frac{q(z|x)}{p(z|x)}\right) \\ &= E\left(\log q(z|x)\right) - E\left(\log p(z|x)\right) \\ &= E\left(\log q(z|x)\right) - E\left(\log p(z,x)\right) + \log p(x) \\ &\propto E_{q_\phi}\left(\log q_\phi(z|x)\right) - E_{q_\phi}\left(\log p(z,x)\right) \end{aligned}$$



WORKING WITH THE ELBO

ELBO:

$$F(\phi, \theta) = -E_{q_\phi}\left(\log q_\phi(z|x)\right) + E_{q_\phi}\left(\log p_\theta(z, x)\right)$$



- Note negation (a convention)
- Also introduction of θ

View this setup as *dimension reduction*:

- $p_\theta(x|z)$ is a *probabilistic decoder*, converting latent code z to observed data x
- $q_\phi(z|x)$ is a *probabilistic encoder*, converting observed data x to latent code z
- Now we must choose our approximating family Q...

Example generative model: neural networks as flexible, expressive function families:

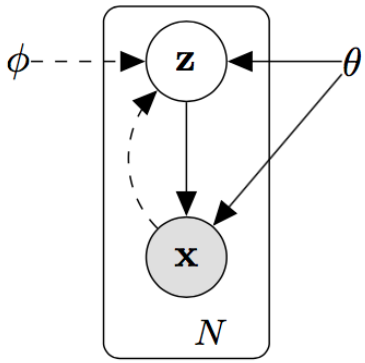
$$p_\theta(x|z) = \mathcal{N}\left(\mu_\phi(z), \sigma^2_\phi(z)\right)$$

- Here both μ_θ and σ_θ (diagonal matrix) are neural networks that map $\mathcal{Z} \to \mathcal{X}$

VARIATIONAL FAMILY Q

ELBO:

$$F(\phi, \theta) = -E_{q_\phi}\left(\log q_\phi(z|x)\right) + E_{q_\phi}\left(\log p_\theta(z, x)\right)$$



- Note negation (a convention)
- Also introduction of θ
- Suppose $z \in \mathbb{R}^d$ and $x \in \mathcal{X}$

Neural networks as flexible, expressive function families (again):

$$q_\phi(z|x) = \mathcal{N}\left(\mu_\phi(x), \sigma^2_\phi(x)\right)$$

- Here both μ_ϕ and σ_ϕ are neural networks that map $\mathcal{X} \to \mathbb{R}^d$
- Perhaps easier to view this from the perspective of a noise variable ϵ:

$$\epsilon \sim \mathcal{N}(0, I_d) \quad \text{and} \quad z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon \quad \to \quad z|x \sim \mathcal{N}\left(\mu_\phi(x), \sigma^2_\phi(x)\right)$$

- This *reparameterization trick* makes it simple to sample from $q_\phi(z|x)$
- Note: this gaussian is one basic choice, but many others are used

STOCHASTIC OPTIMIZATION OF ϕ

We still have the issue of calculating (and differentiating!) these expectations:

$$\arg\max_{\phi} F(\phi, \theta) = \arg\max_{\phi} -E_{q_\phi}\left(\log q_\phi(z|x)\right) + E_{q_\phi}\left(\log p_\theta(z, x)\right)$$

Turn to stochastic optimization and mini-batch gradient descent:

- Draw a noise minibatch $\epsilon_1, \ldots, \epsilon_M$ iid from $\mathcal{N}(0, I)$
- Draw a data minibatch $x_1, \ldots, x_M$ from the dataset
- Compute $z_m = \mu_\phi(x_m) + \sigma_\phi(x_m) \odot \epsilon_m$
- Approximate objective:

$$\hat{F}(\phi) = -\frac{1}{M}\sum_{m=1}^{M} \log q_\phi(z_m|x_m) + \frac{1}{M}\sum_{m=1}^{M} \log p_\theta(z_m, x_m)$$

- Take its gradient and follow SGD (Adam, etc.) in the usual way until an optima is reached

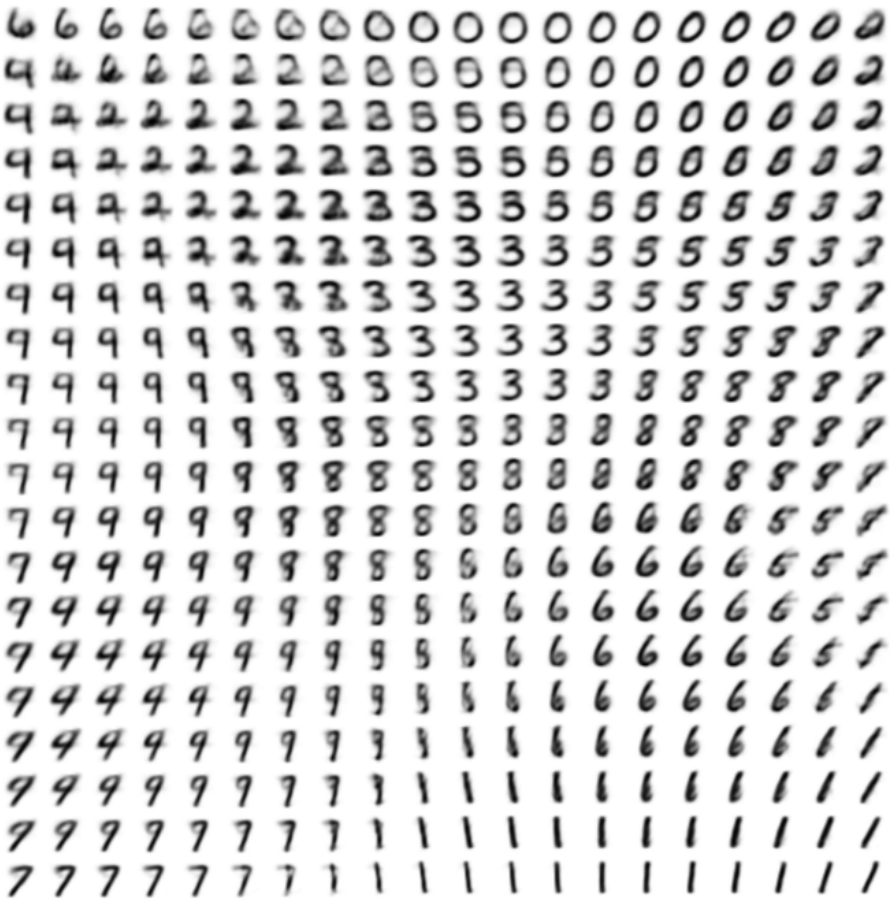
Optimizing this objective:

- Learns a posterior approximation $q_\phi(z|x)$ that can be queried for any data point x
- can be done with a prescribed model $p(x, z)$ to do inference
- or can also take gradients in θ and learn $p_\theta \rightarrow$ dimension reduction/autoencoding
- We call this general approach variational autoencoding (VAE)

VARIATIONAL AUTOENCODER IN ACTION

Learn the autoencoder and then:

- Choose a point z_i in latent space (not drawing from the posterior!)
- Decode this point with $x_i \sim p_\theta(x_i|z_i)$:

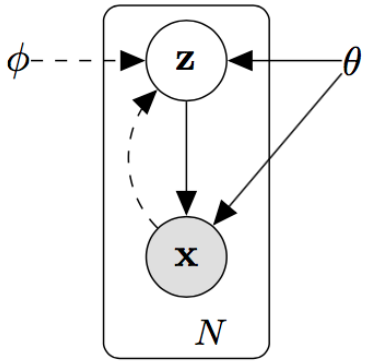


Learns a manifold of simple images and how to generate...

From VAE to GAN

Variational autoencoders:

- ...are designed to do inference
- ...are seen as dimension reduction
- can generate, but that is not their specific design...



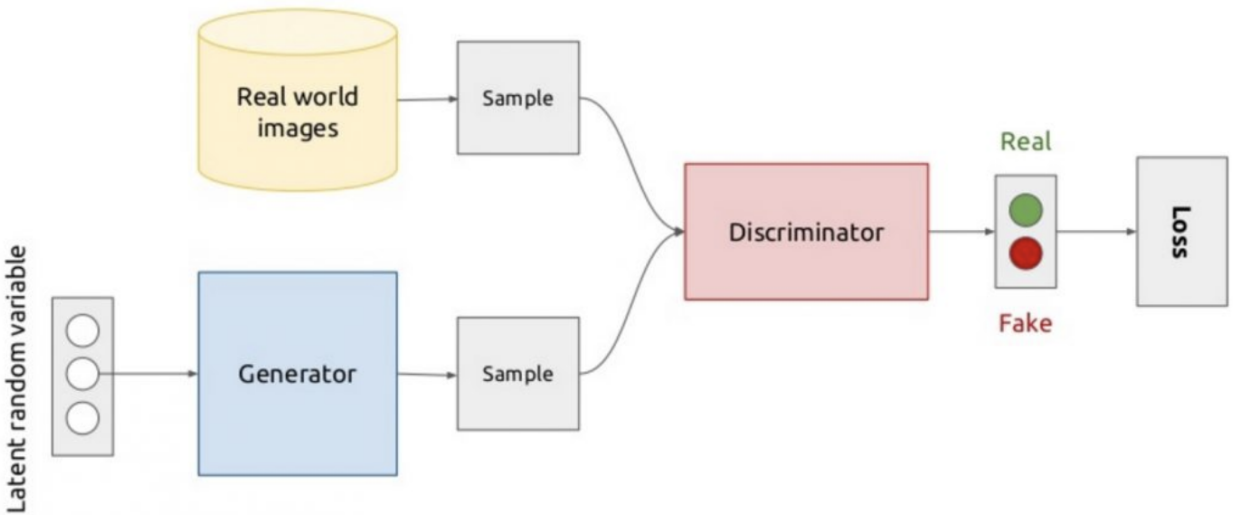
A useful analogy for the idea of directly solving the data generation problem:

https://www.secretservice.gov/data/KnowYourMoney.pdf

GENERATIVE ADVERSARIAL NETWORKS

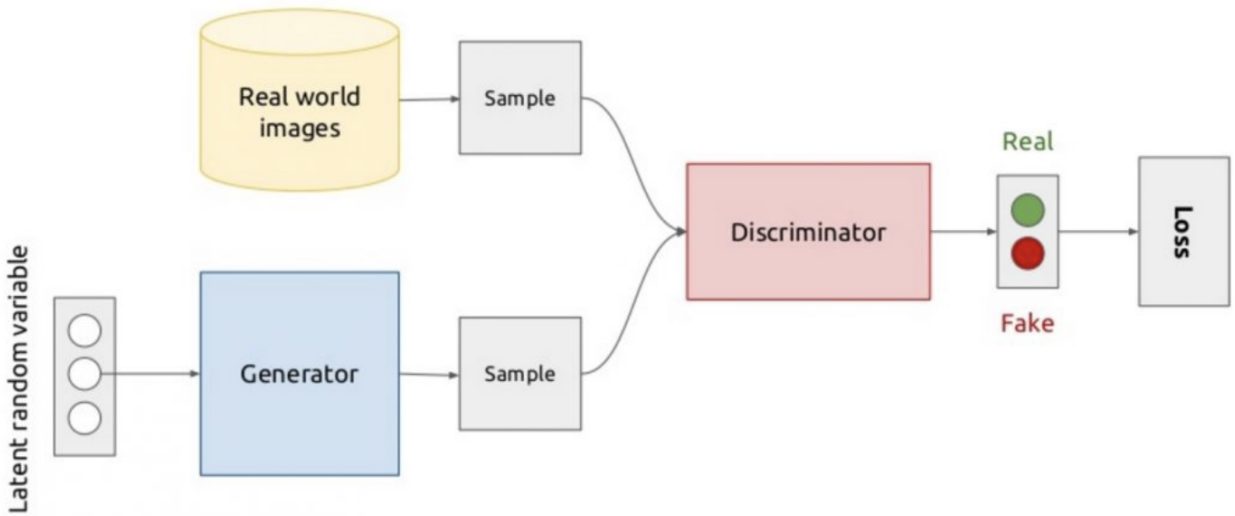
From a deep learning perspective:

- True data samples $x_i^D \sim p_{data}(x)$... (minibatch) draws from the training set
- The latent code $z_i \sim \mathcal{N}(0, I)$
- The *generator* neural network $x_i^G = G_{\phi_G}(z_i)$
- The *discriminator* neural network $D_{\phi_D}(x_i) \rightarrow [0, 1]$



[image from http://cognitivechaos.com/understanding-generative-adversarial-networks/]

- The discriminator classifies fake vs real images
- The generator adapts to fool the discriminator
- This two-player game is repeated...

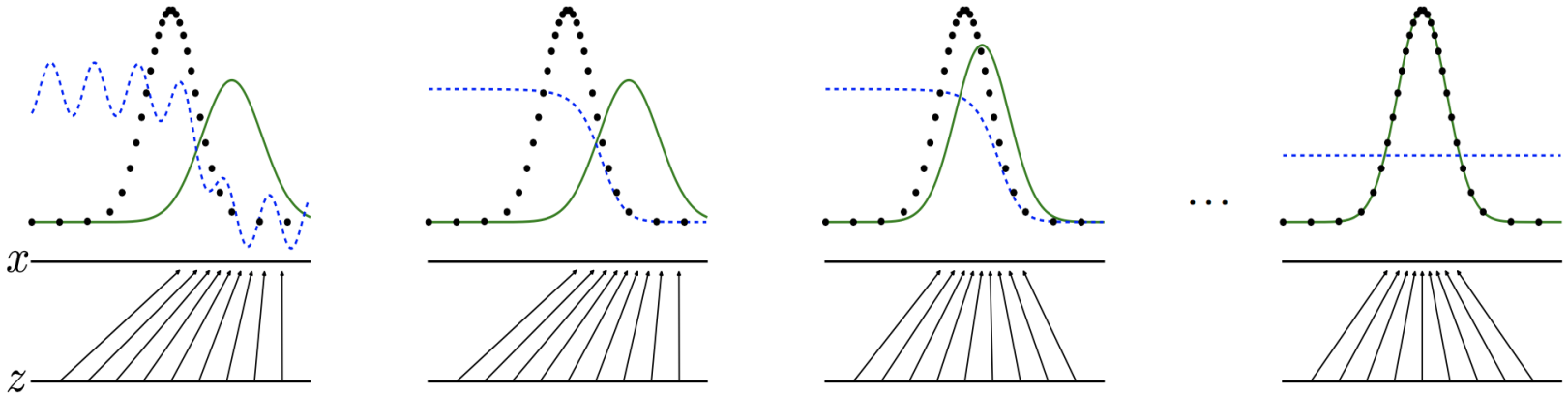


Specify the following objective:

$$\min_{\phi_G} \max_{\phi_D} \left[E_{x \sim p_{data}} \left(\log D_{\phi_D}(x) \right) + E_{z \sim p(z)} \left(\log \left(1 - D_{\phi_D} \left(G_{\phi_G}(z) \right) \right) \right) \right]$$

- $D_{\phi_d}(x_i)$ gives the probability ($\in [0, 1]$) that x_i^D is genuine (from data distribution)
- $1 - D_{\phi_D} \left(G_{\phi_G}(z_i) \right)$ gives the probability that x_i^G is *fake*
- $\min_{\phi_G}$ attempts to minimize the probability of being caught as a fake
- $\max_{\phi_D}$ attempts to maximize discriminability (reals $\uparrow$, fakes $\downarrow$)...

$$\min_{\phi_G} \max_{\phi_D} \left[E_{x \sim p_{data}} \left(\log D_{\phi_D}(x) \right) + E_{z \sim p(z)} \left(\log \left(1 - D_{\phi_D} \left(G_{\phi_G}(z) \right) \right) \right) \right]$$



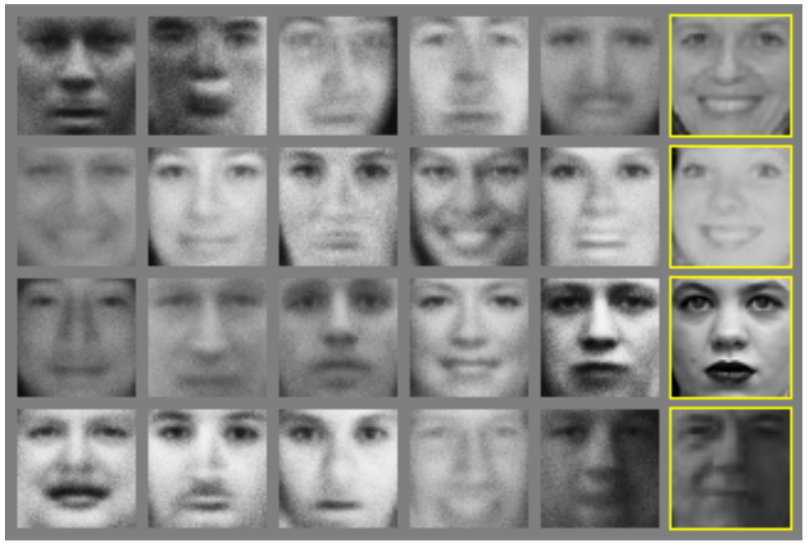
Here:

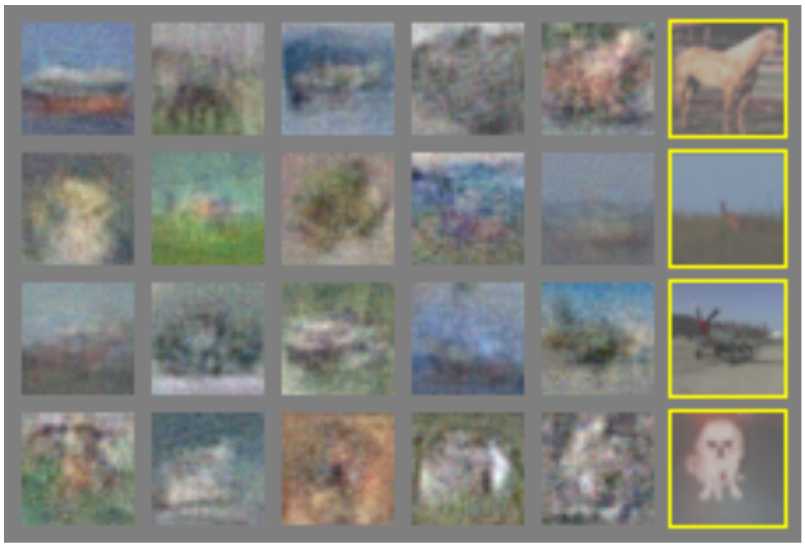
- Discriminator $D(x)$ (blue); generative distribution $p_G(x)$ (green); true $p_{data}(x)$ (black)
- Second panel: if arbitrarily expressive, $\max_D$ optimizes to $D_{\phi_D}(x) = \frac{p_{data}(x)}{p_{data}(x)+p_G(x)}$
- If everything works, eventually p_G is indistinguishable from p_{data}...

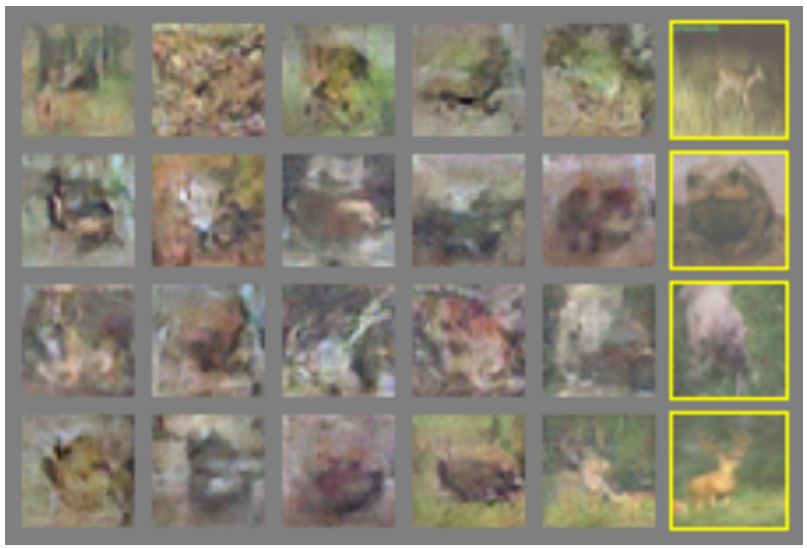
Note:

- Do not take this objective/optima as absolute truth: original idea, several times updated
- Theory is starting to appear...
- Discuss *mode collapse* and learning/generating the training set

GAN IN ACTION







(right column images are nearest neighbor training points)

GAN IN ACTION

Is this good? What could we do with it if it were?

GAN: LATEST AND GREATEST

GAN: LATEST AND GREATEST

https://arxiv.org/abs/1710.10196

GAN: LATEST AND GREATEST

SOME CONTEXT

Implicit probabilistic modeling with neural networks is an exciting area of development:

- Heavily demonstrated in the computer vision space
- Expanding to many areas of statistical modeling
- ...including my own research:
  - dynamical systems / state space models [https://arxiv.org/abs/1511.07367]
  - maximum entropy modeling [https://arxiv.org/abs/1701.03504]
  - pathologies of variational autoencoders [https://arxiv.org/abs/1907.06845]
- And much more: [http://stat.columbia.edu/~cunningham/teaching/GR8201/]

That said, serious skepticism about IPM (and neural networks generally) still exists:

- Serious concerns about generalization in GAN [https://arxiv.org/abs/1703.00573]
- Not even clear why neural networks work well [https://arxiv.org/abs/1611.03530]
- Heard last year at a major conference: “Deep learning is unrigorous alchemy”...

Deep learning, and in particular the applications and algorithms we have learned here, are both very exciting and not entirely understood. Have fun and be thoughtful!