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

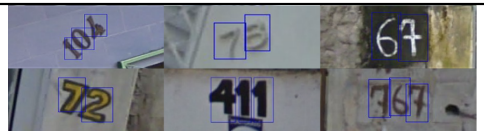
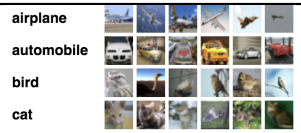
The four catalysts of the AI explosion

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

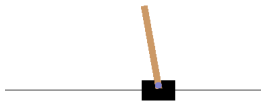
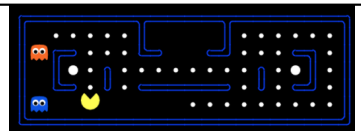
We will study some neural network families $\mathcal{F}$. While neural networks are powerful, there is nothing magical or fundamentally different than what you already know.

CATALYST 1: DATA

Computer Vision

SVHN	CIFAR10	ImageNet	...
			...

Reinforcement Learning

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

Natural Language Processing

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

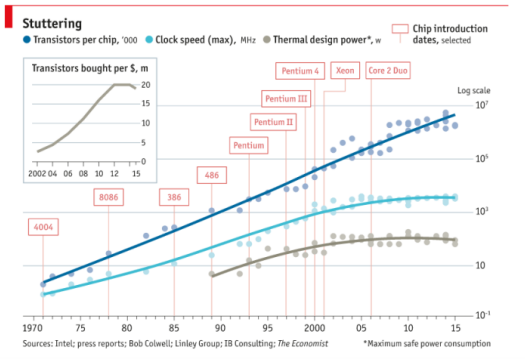
See https://github.com/niderhoff/nlp-datasets

And so much more...

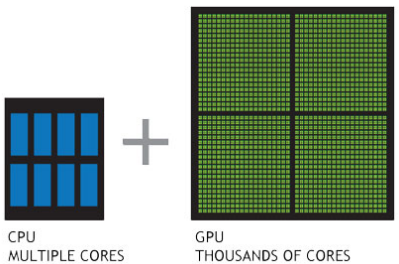
- `https://www.data.gov/`
- `https://opendata.cityofnewyork.us/`
- `https://github.com/caesar0301/awesome-public-datasets`
- `https://data.world`
- ...

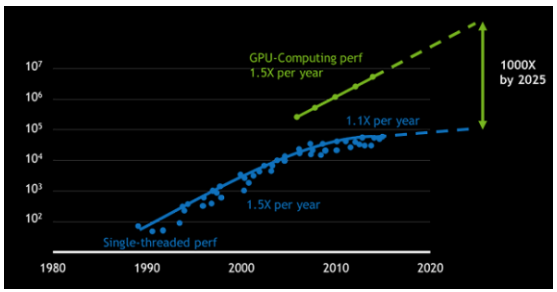
CATALYST 2: COMPUTATIONAL POWER

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

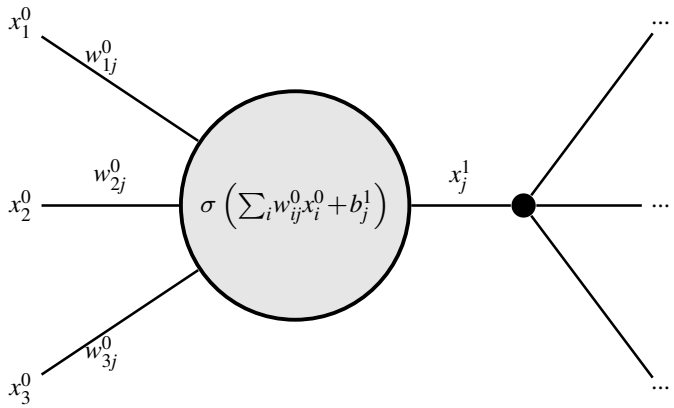


GPUs have accelerated this trend, especially important for ML-relevant computation

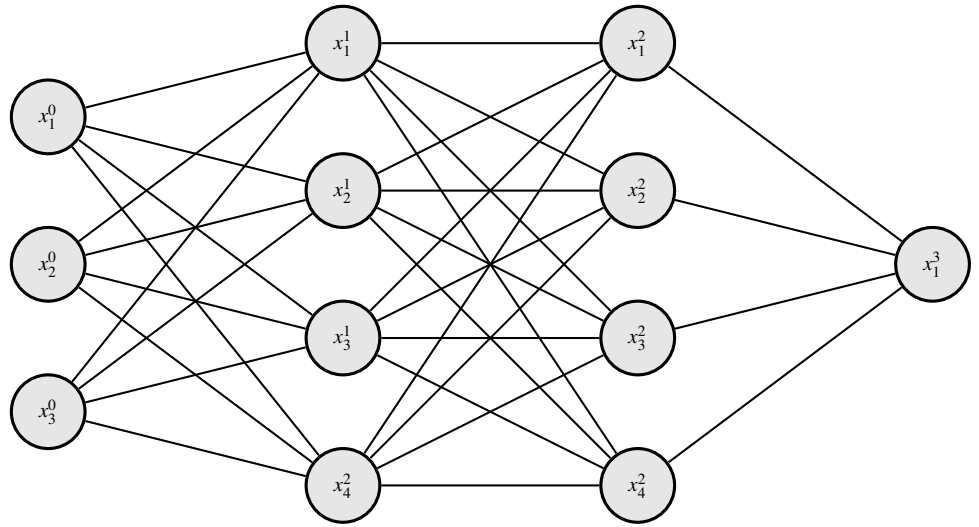




Cloud computing has made this even easier (abstracting away IT and system ops)



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

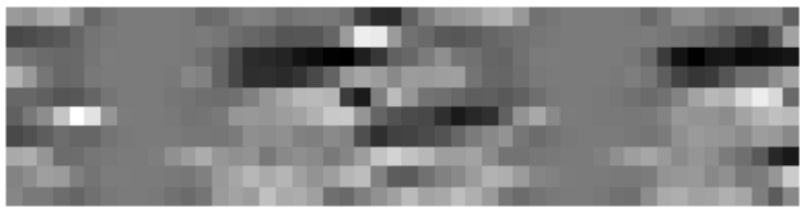
Learning/Training is then minimizing an empirical risk over the parameter set

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

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Logistic Regression

x



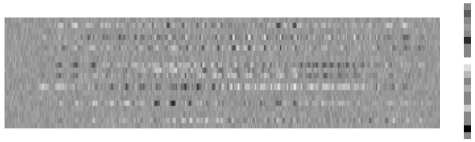
W

b

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

Neural Network

x



W_1

b_1

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

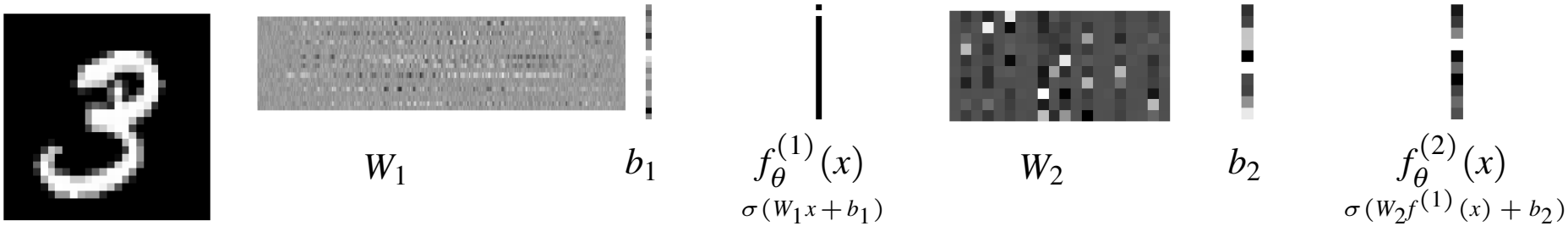
W_2

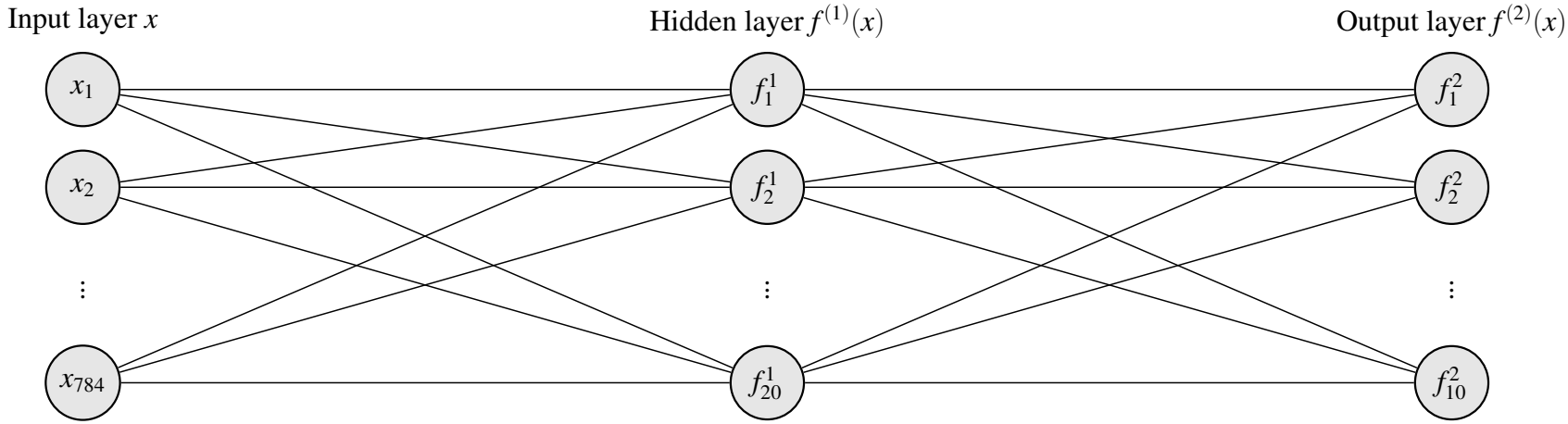
b_2

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

EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Neural Network





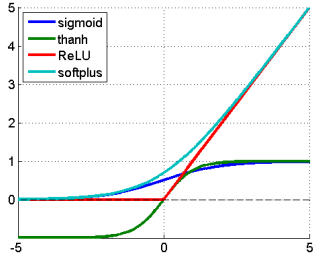
Cascade layers for any amount of depth and complexity!

Naive conclusion: deep learning is easy...

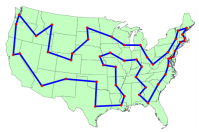
...Deep learning is hard

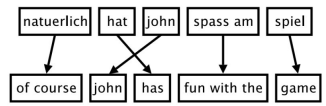
- How do I choose $\left|f^{(1)}\right|$, the number of *units* in the hidden layers?
- How do I choose L, the number of *layers*?
- How do I choose the *activation function* $\sigma(\cdot)$?

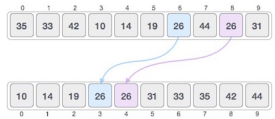
sigmoid	tanh	relu	softplus	softmax	...
$\frac{1}{1+e^{-x}}$	$\frac{e^{x}-e^{-x}}{e^{x}-e^{-x}}$	$\max(0,x)$	$\log\left(1+e^{x}\right)$	$\frac{e^{x_i}}{\sum_k e^{x_k}}$	...



- 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

TensorFlow™ Keras theano torch Caffe ...

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

- Automatic differentiation

```
In [119]: # how to predict label from data
          y_model = tf.nn.softmax(tf.matmul(x,W) + b)
          # the objective function
          cross_ent = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_model), reduction_indices=[1]))
          # the cost function to be optimized
          train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_ent)
          # performance quantification
          correct_pred = tf.equal(tf.argmax(y_model,1), tf.argmax(y,1))
          accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

- Stochastic optimization

```
In [165]: with tf.Session() as sess:
              sess.run(tf.global_variables_initializer())
              # train model
              for i in range(1001):
                  train_step.run(feed_dict={x: X_train, y: y_train})
                  # print diagnostics
```

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!

```
In [119]: # how to predict label from data
          y_model = tf.nn.softmax(tf.matmul(x,W) + b)
          # the objective function
          cross_ent = tf.reduce_mean(-tf.reduce_sum(y*tf.log(y_model), reduction_indices=[1]))
          # the cost function to be optimized
          train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_ent)
          # performance quantification
          correct_pred = tf.equal(tf.argmax(y_model,1), tf.argmax(y,1))
          accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

Somehow tensorflow took the gradients under the hood

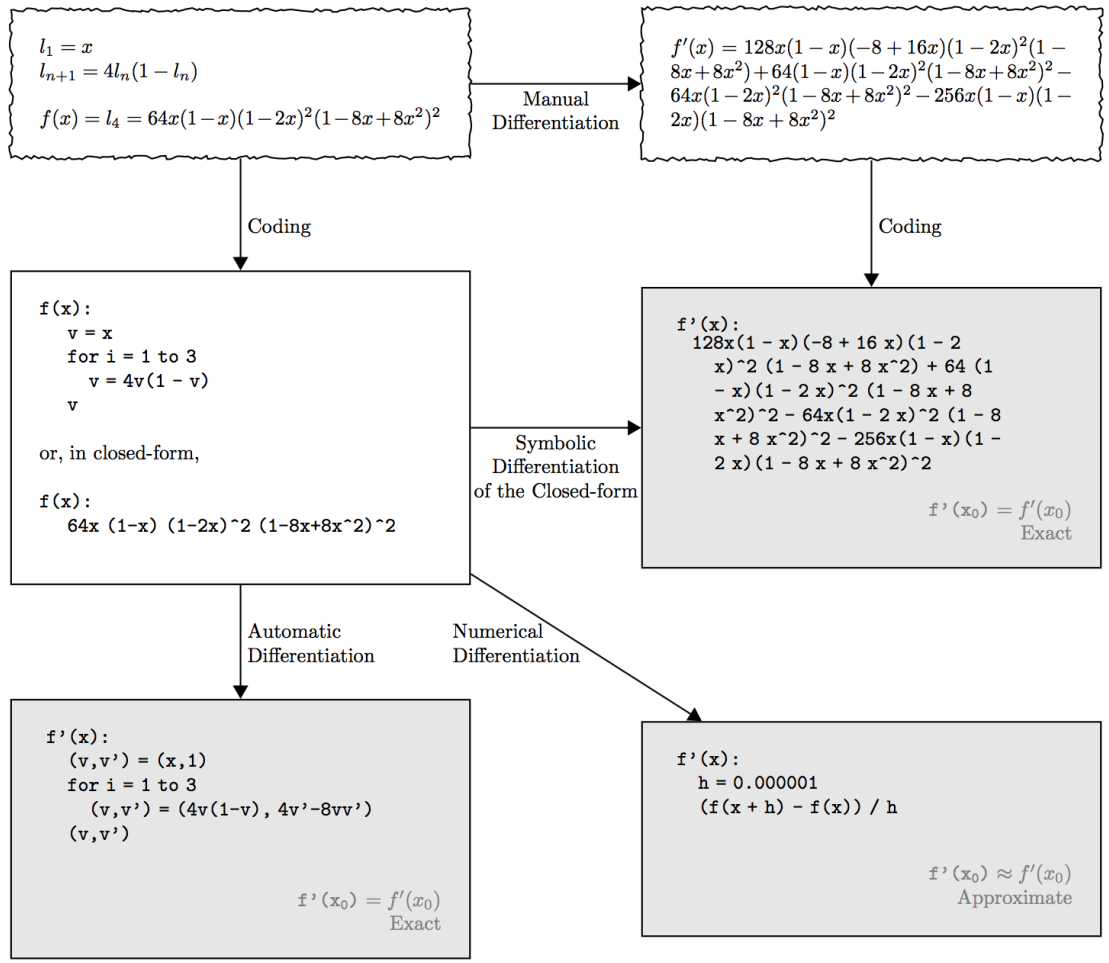
DIFFERENTIATION

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



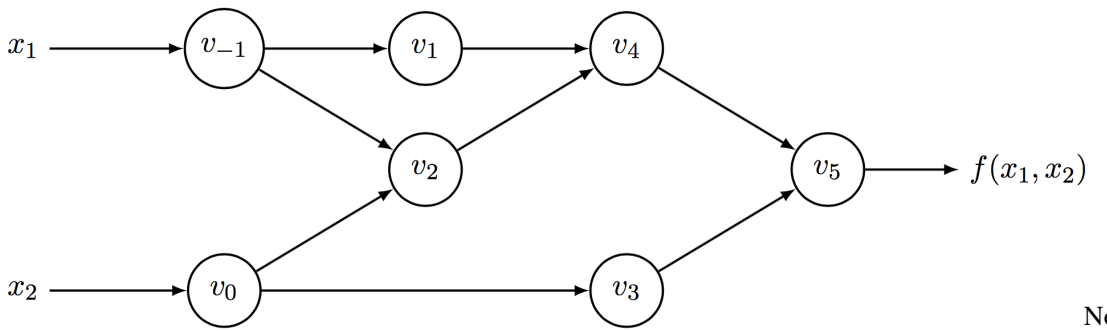
[Baydin et al (2015) JMLR... note the for loop!]

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.

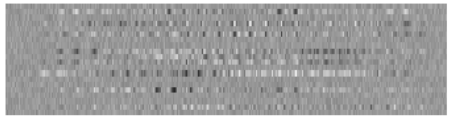
Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
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	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

[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

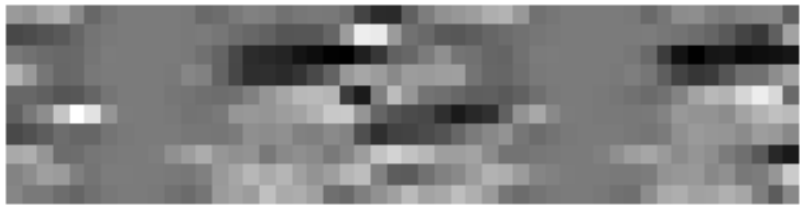
- Audodiff is old and many times reinvented; yes it's just the chain rule.
- Machine learning was embarrassingly slow to adopt autodiff. Now it's pervasive.
- Can I just forget calculus? No! ...but also (sort of) Yes!

TOOLS: STOCHASTIC OPTIMIZATION

EXAMPLE: LOGISTIC REGRESSION $\rightarrow$ NEURAL NETWORKS

Logistic Regression

x



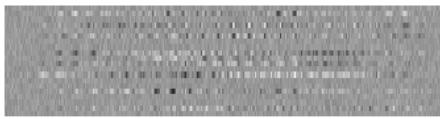
W

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

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

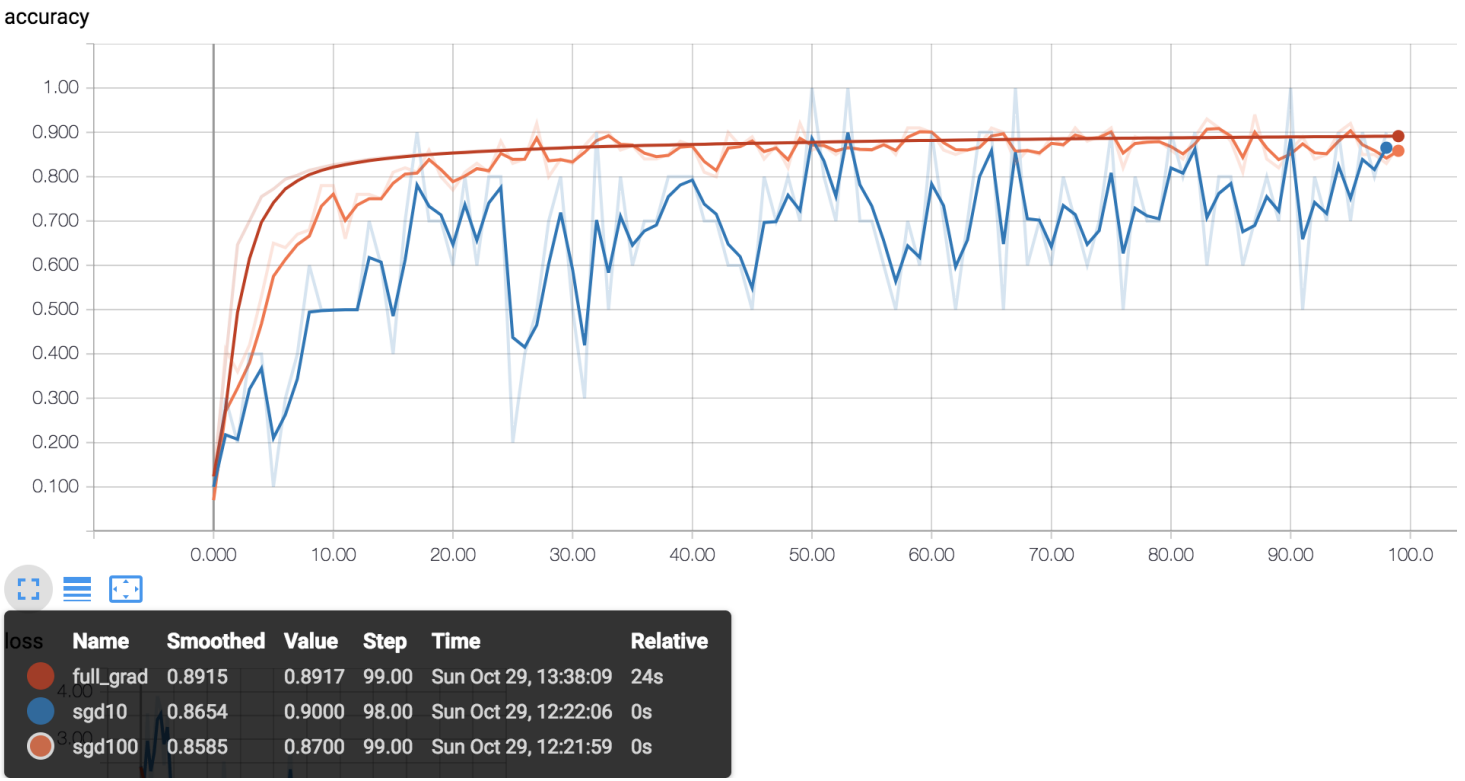
Concerns:

- 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 $x_{i_1}, \ldots, x_{i_M}$

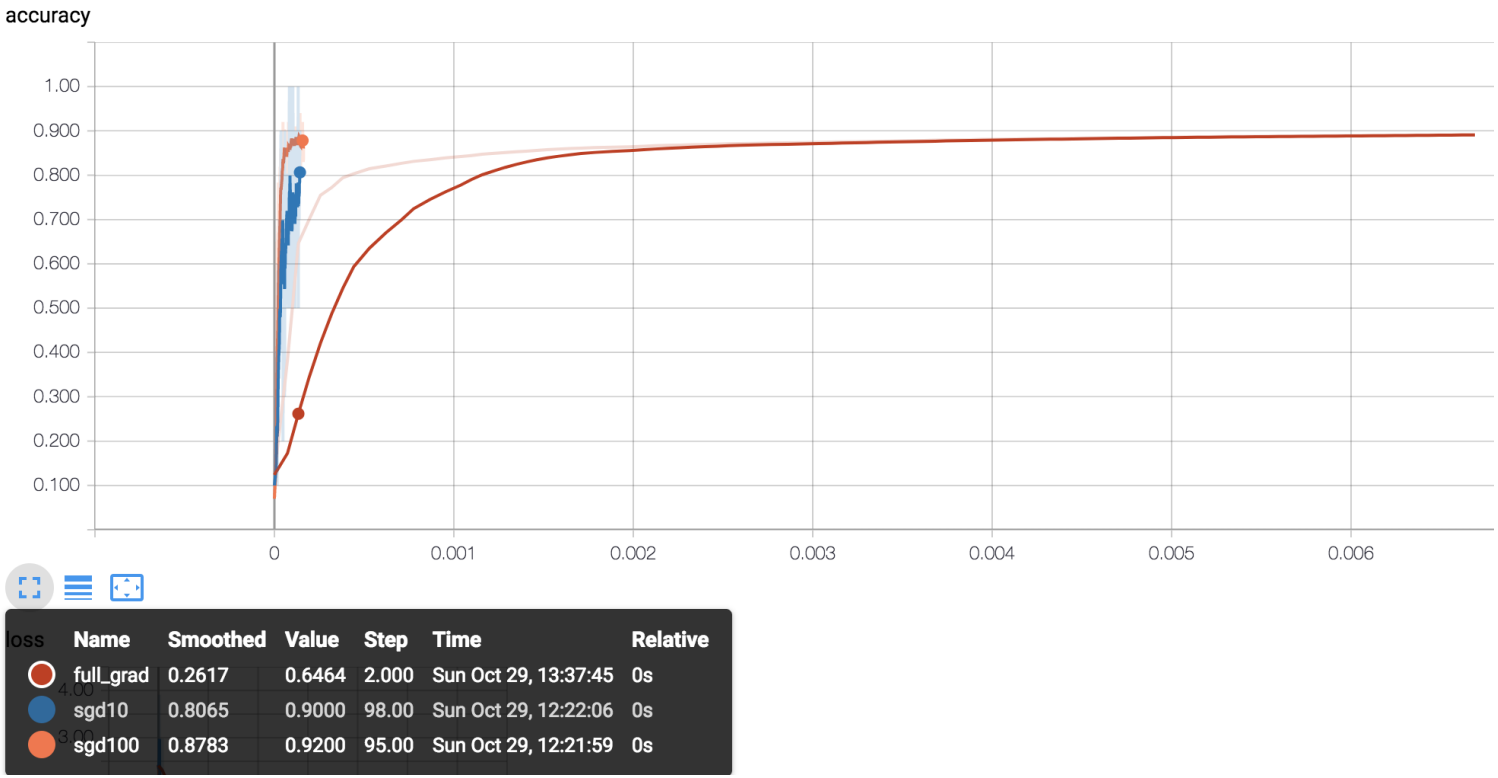
$$\theta^{(k+1)} \leftarrow \theta^{(k)} - \alpha_k \frac{1}{M} \sum_{m=1}^{M} \nabla_\theta L\left(y_{i_m}, f_\theta\left(x_{i_m}\right)\right)$$



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!



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...

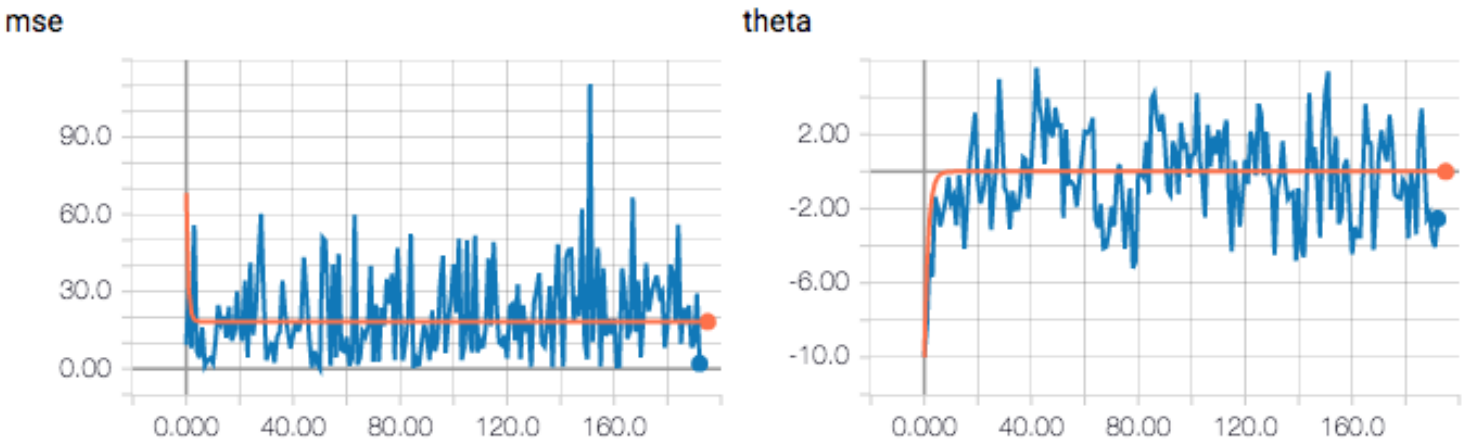
DANGER! SGD REQUIRES CARE

Use SGD to solve this problem:

- Data $\{x_1, ..., x_{21}\} = \{-10.0, -9.0, ..., 0.0, ..., 9.0, 10.0\}$
- 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.

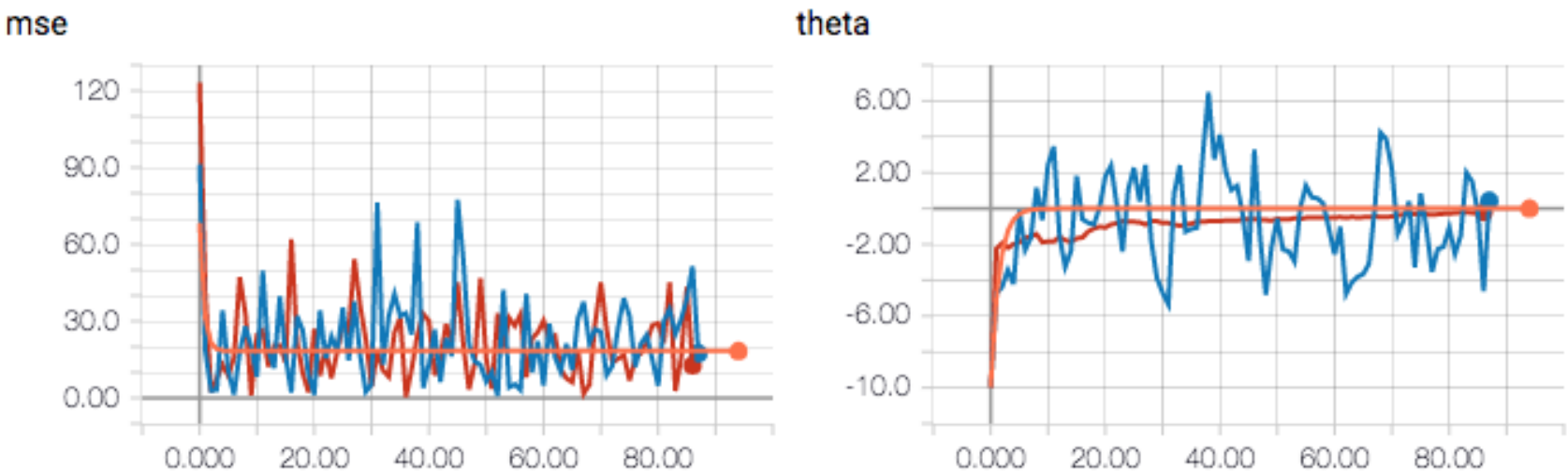
ROBBINS-MONRO

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/...)

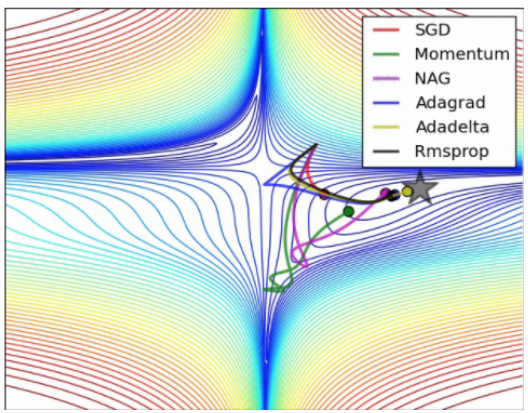


Image 5: SGD optimization on loss surface contours

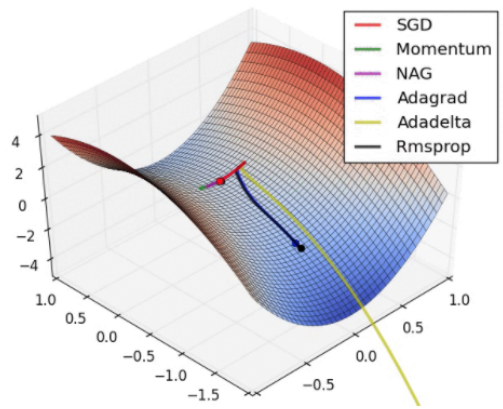


Image 6: SGD optimization on saddle point

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/

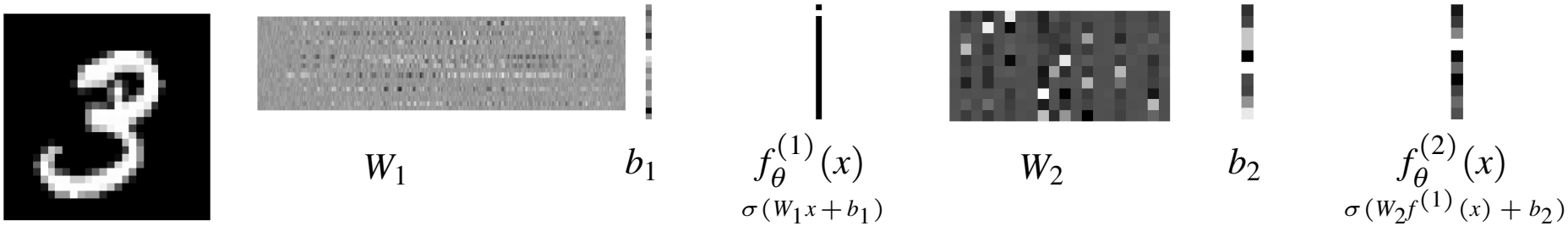
Does this feel abrupt or unsatisfying? It should!

- 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

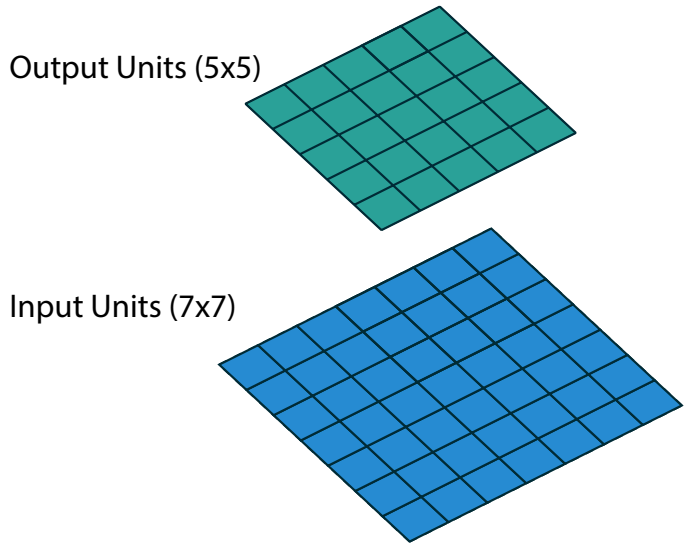
Opportunity:

- 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...

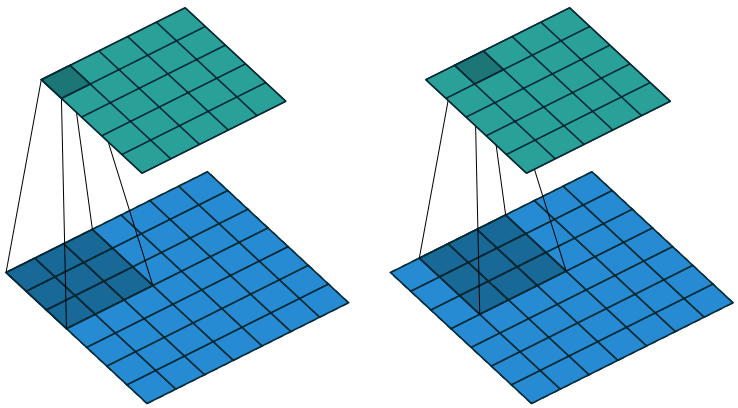
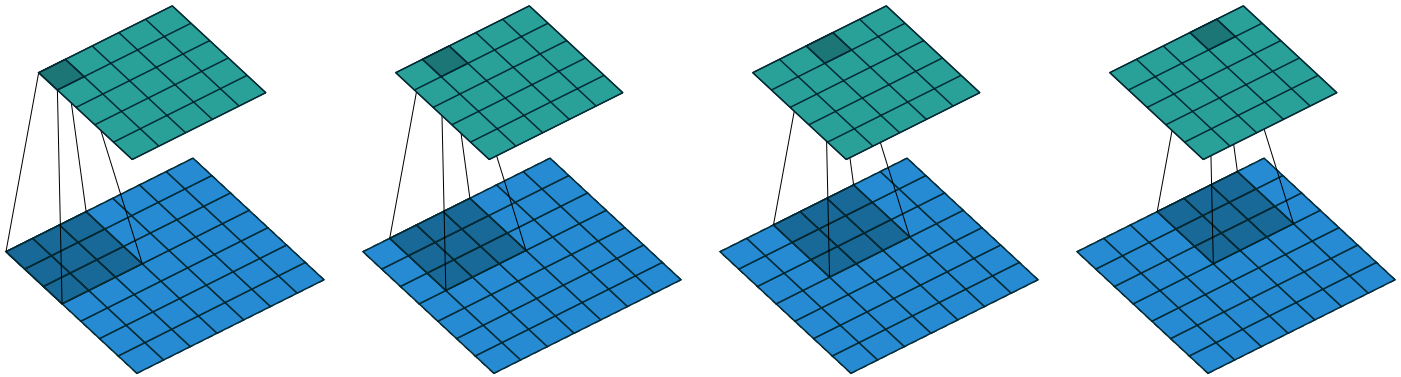


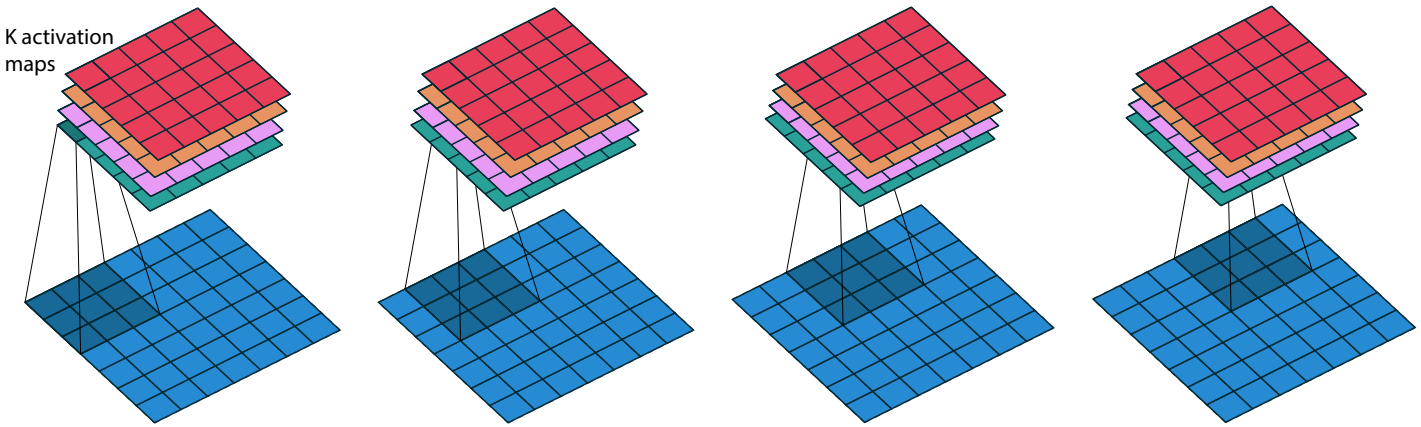
Image credit for all of these and the following: `https://github.com/vdumoulin/conv_arithmetic`

CONVOLUTIONAL LAYER

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



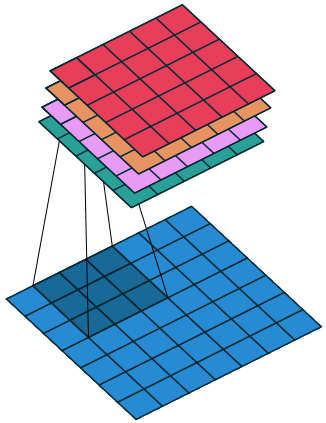
Now use multiple filters (below $K = 4$), producing multiple *activation maps* (each 5×5)



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

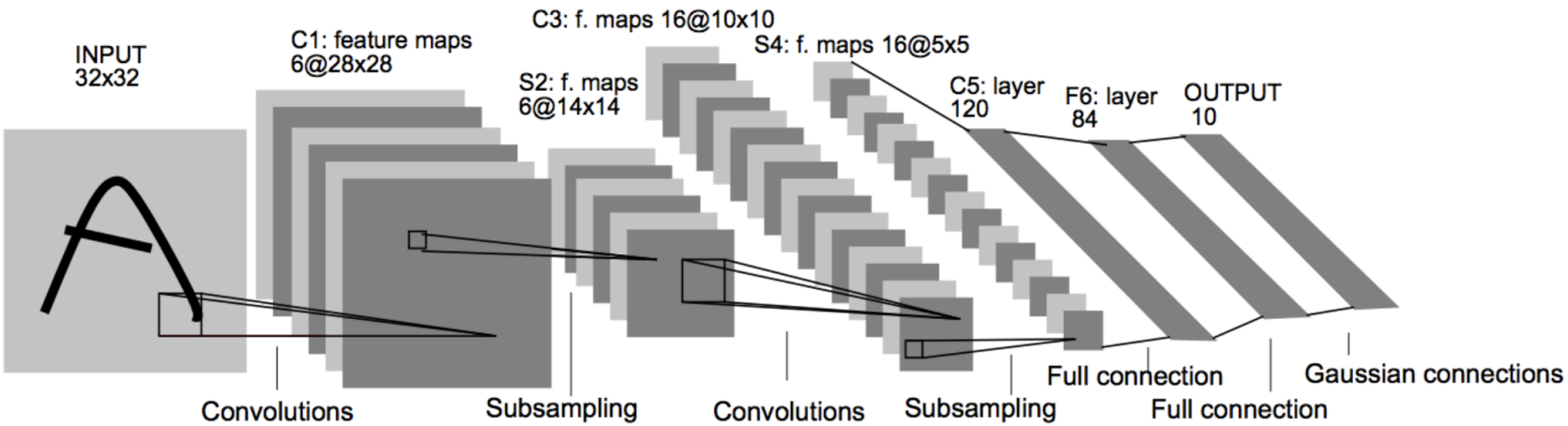
Our/tf convention for 2D convolution: filter width $\times$ filter height $\times$ input depth $\times$ output depth.

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:



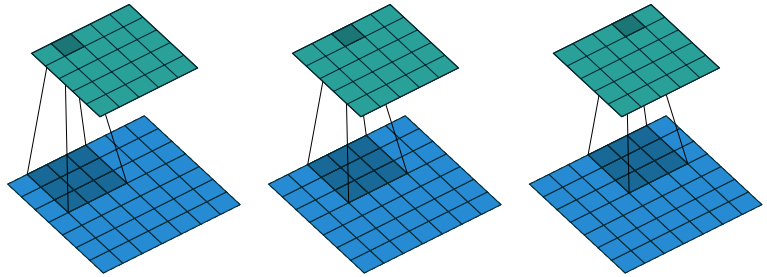
[LeCun et al 1998]

- 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:

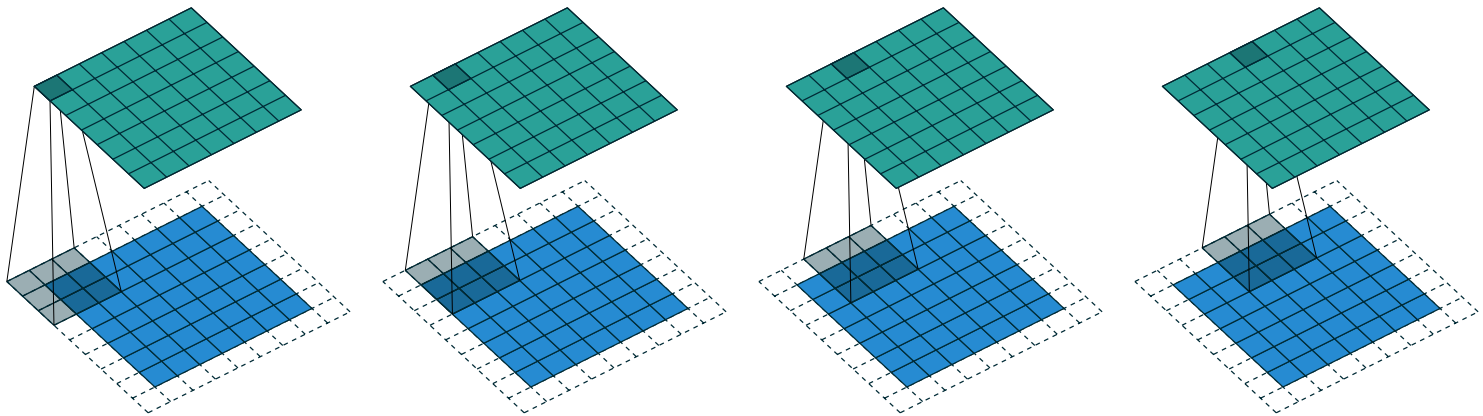
- Filtering reduces spatial extent of activation map
- Edge pixels/activations are less frequently seen
- (Note these can also be benefits)



Zero Padding:

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

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

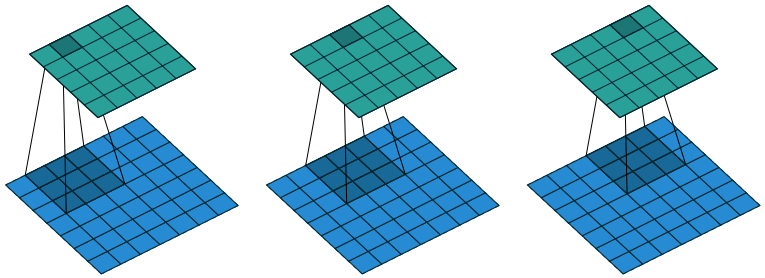


Note: one can zero-pad more/less/asymetrically/otherwise, with varied problem-specific effects

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

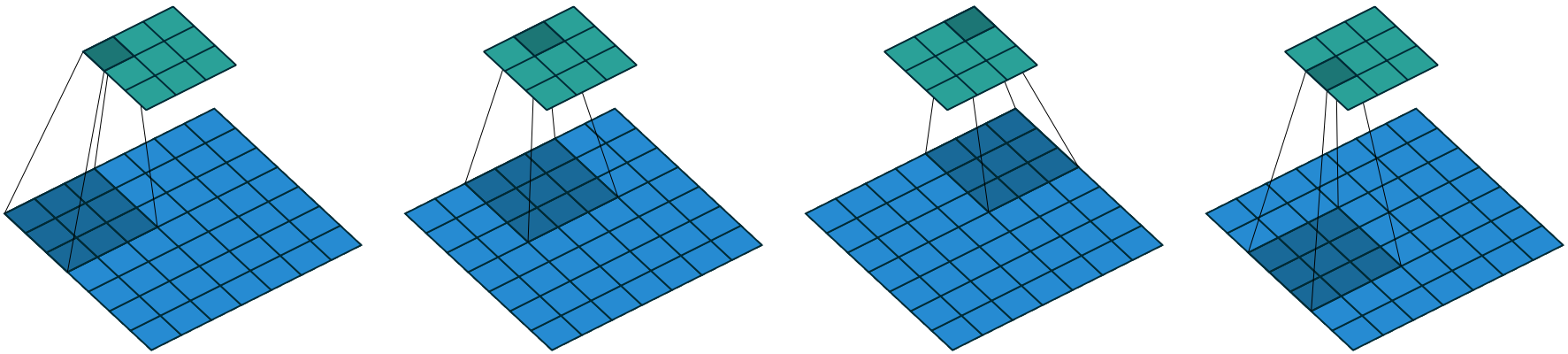


Stride:

- 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.

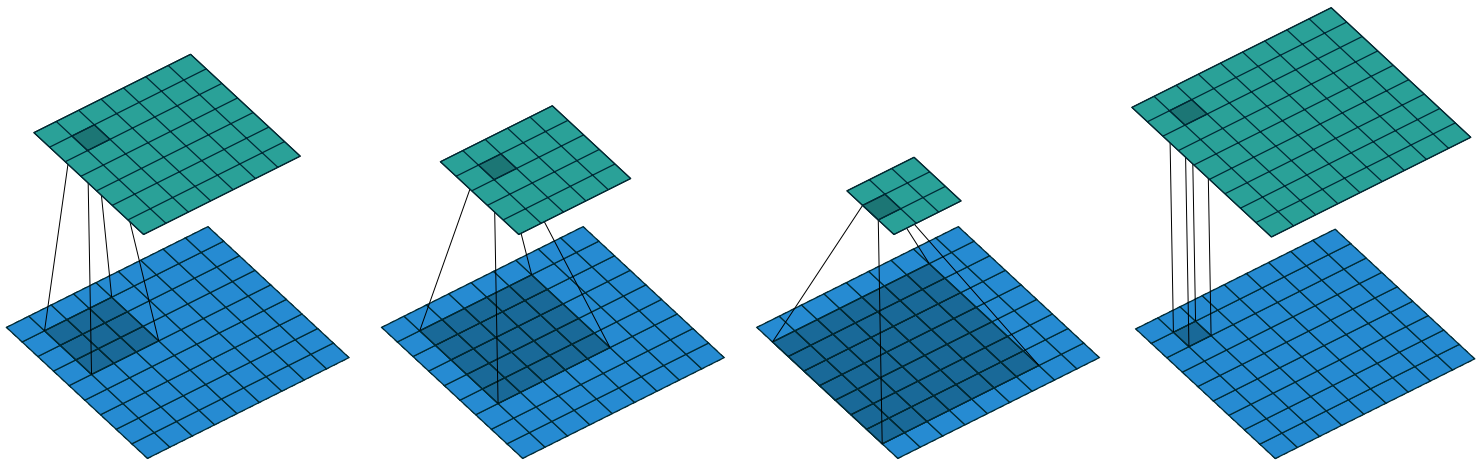


Note: striding and zero-padding give design flexibility and balance each other

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

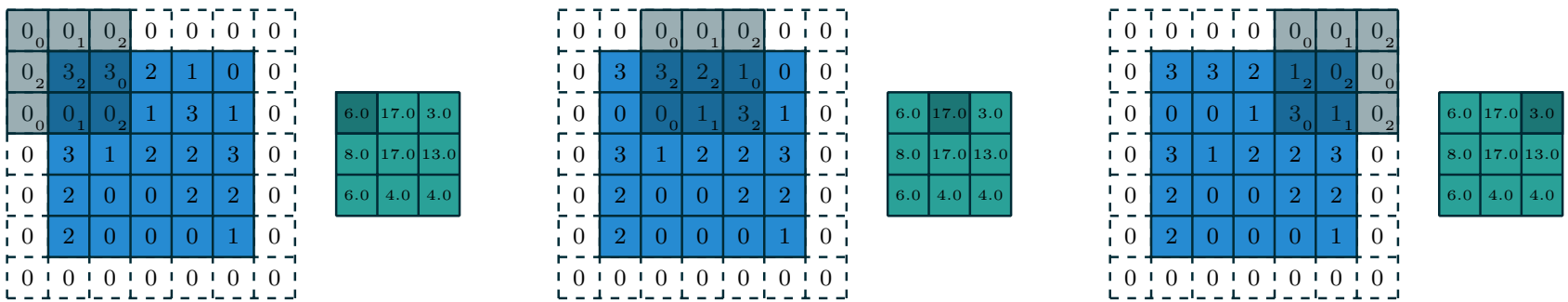
PUTTING THESE ALL TOGETHER

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`

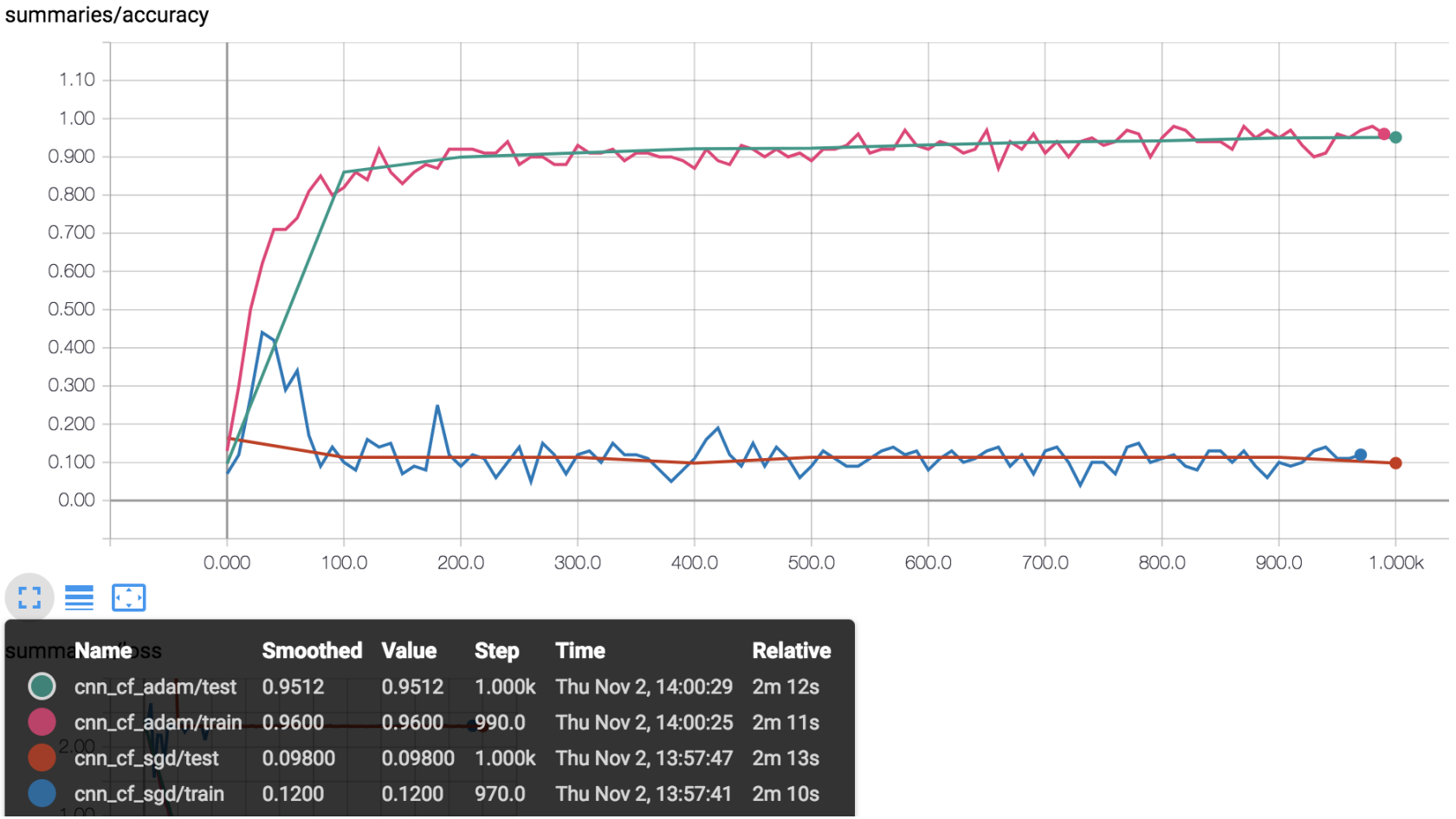
    ...or equivalently `tf.losses.softmax_cross_entropy`
  - `tf.nn.sparse_softmax_cross_entropy_with_logits`

    ...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

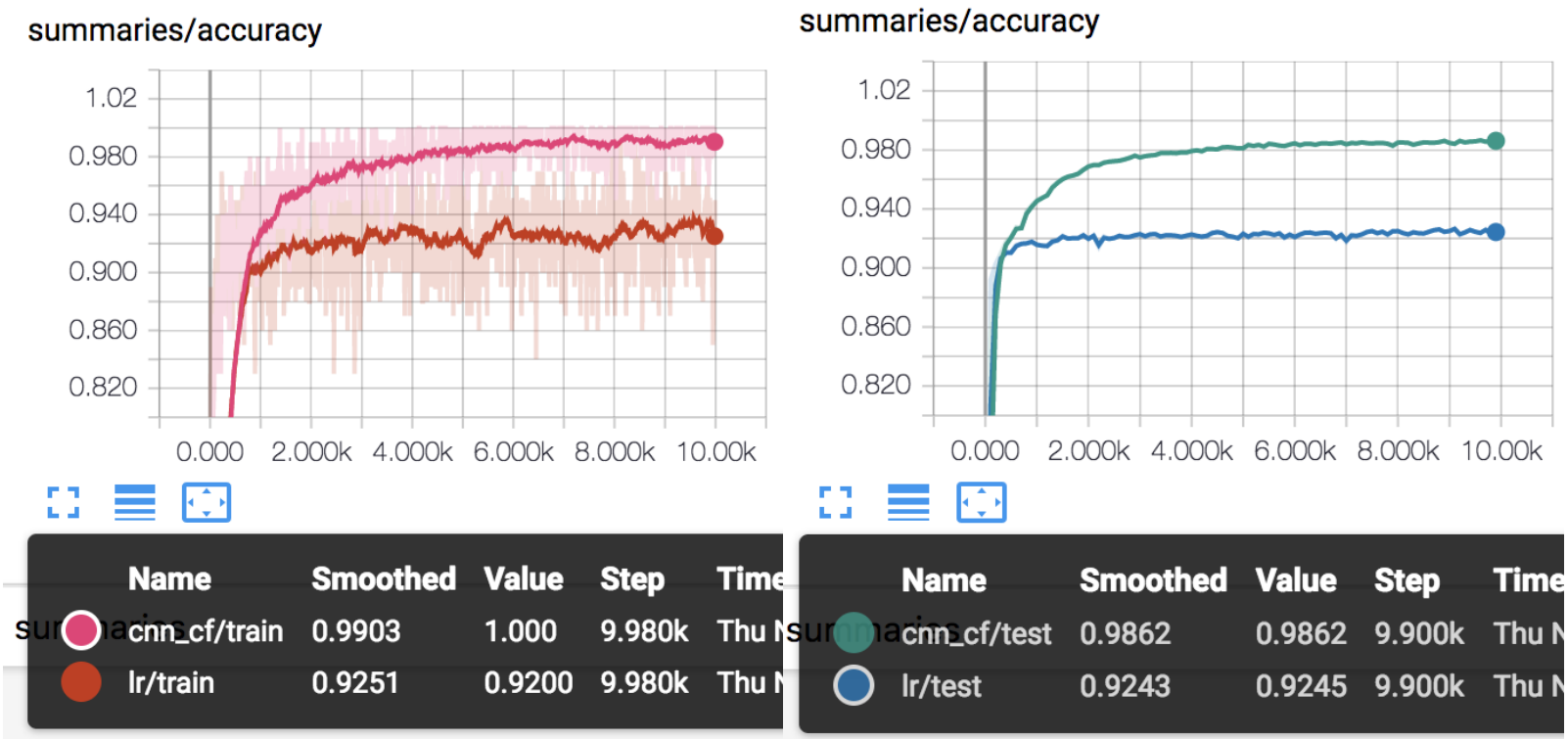
Consider different SGD variants



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

PROGRESS WITH cnn_cf

Training and Test



Questions

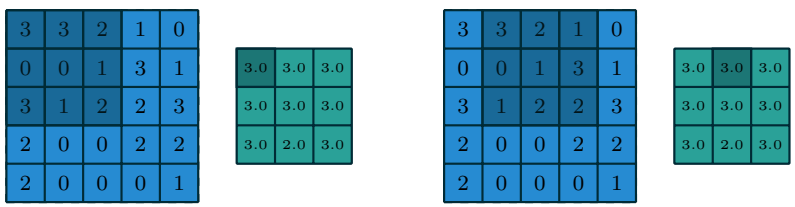
- Why is test/train nonsmooth/smooth?
- How do I set up tensorboard summaries for train and test?
- Will we do better if we make this network more complicated/deeper?
- Am I concerned by a $\approx 0.4\%$ difference between train and test?

TRICKS OF THE TRADE: POOLING

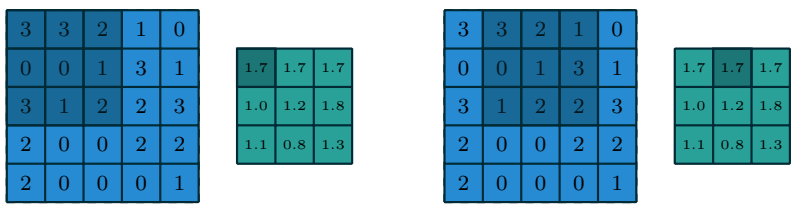
Idea

- 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`

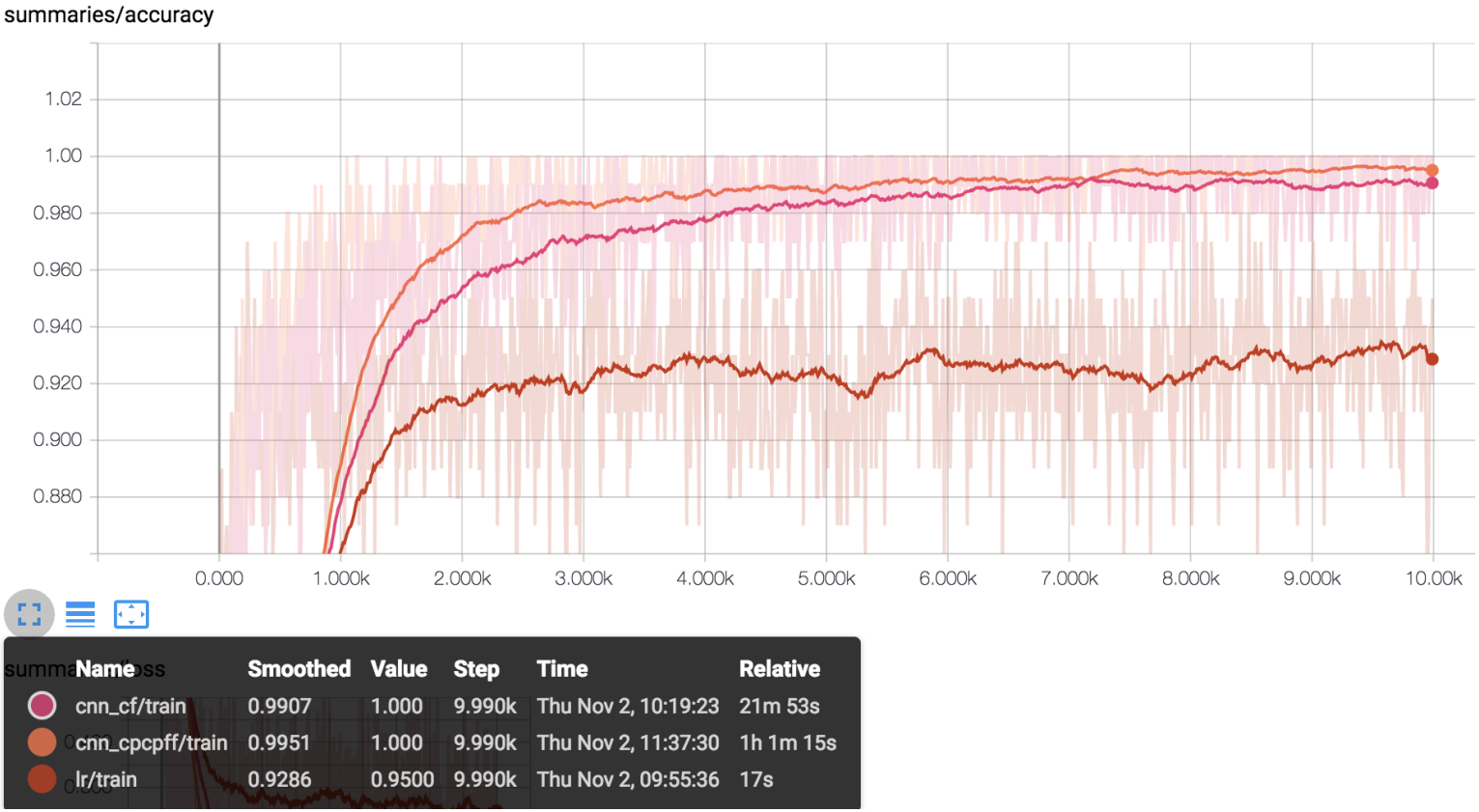
```
elif model_type=='cnn_cpcpff':
    # 2 layer cnn, similar architecture to tensorflow's deep mnist tutorial, so you can compare
    n1 = 32
    n2 = 64
    n3 = 1024
    x_image = tf.reshape(x, [-1,28,28,1]) # batch, then width, height, channels
    # cnn layer 1
    W_conv1 = tf.get_variable('W_conv1', 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

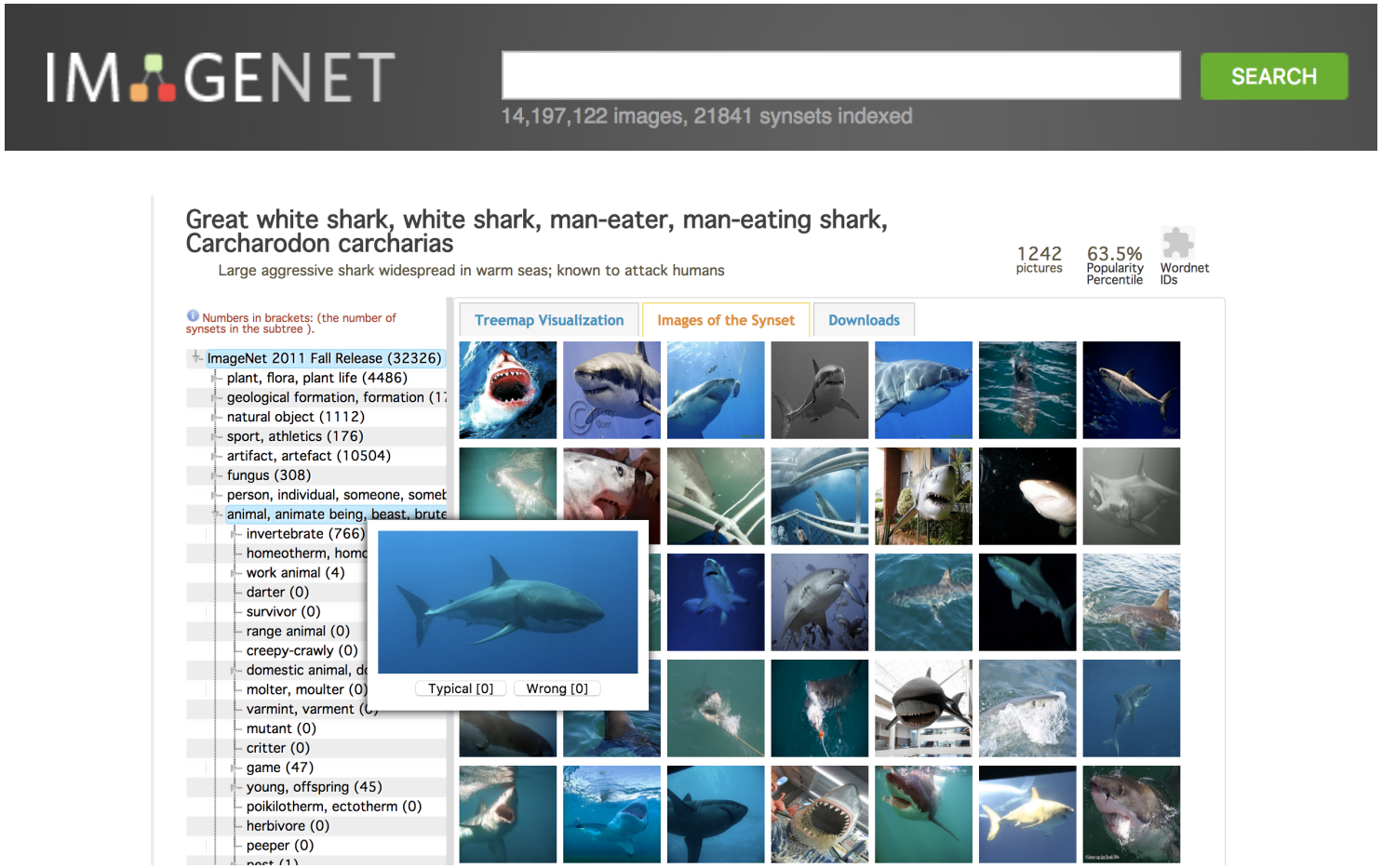


Worth it?

- Better, but not much better.
- More costly

This story will change with more complex datasets...

The best large-scale vision dataset available

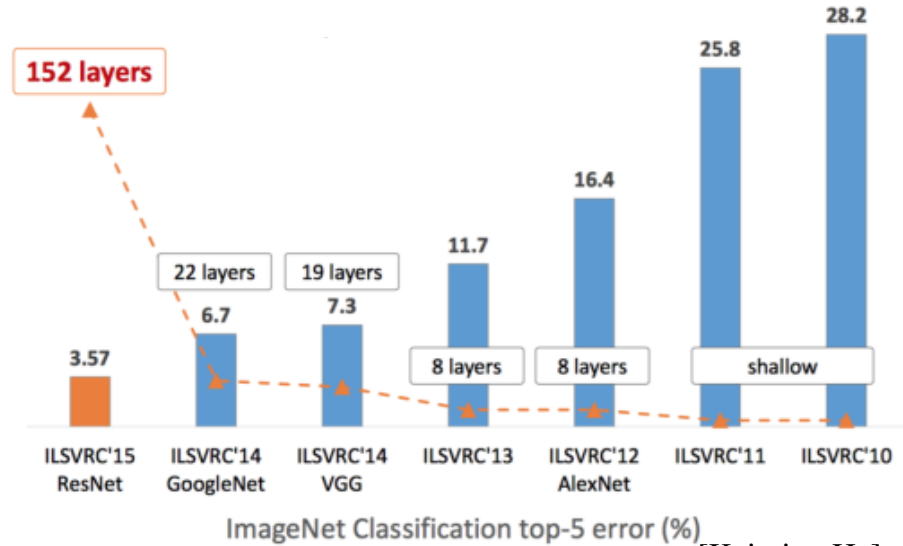


Note also that, in many images, bounding boxes are now provided

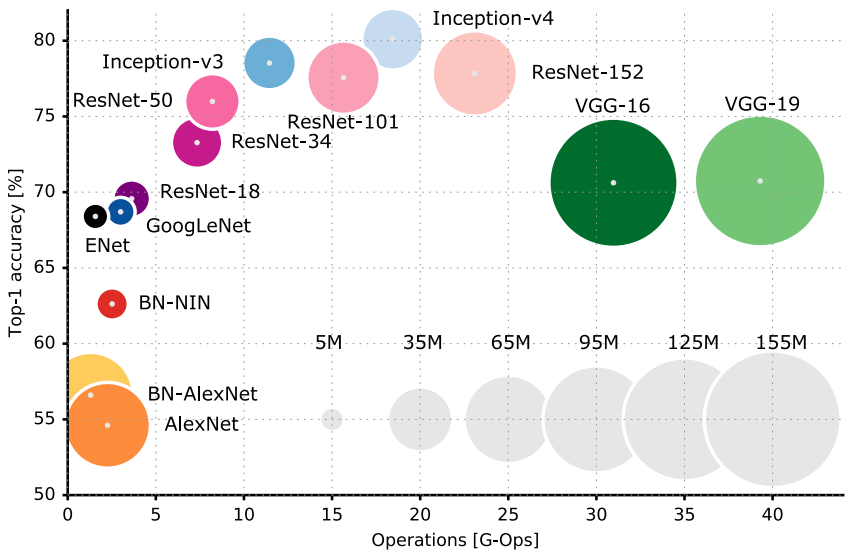
IMAGENET CHALLENGE

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...

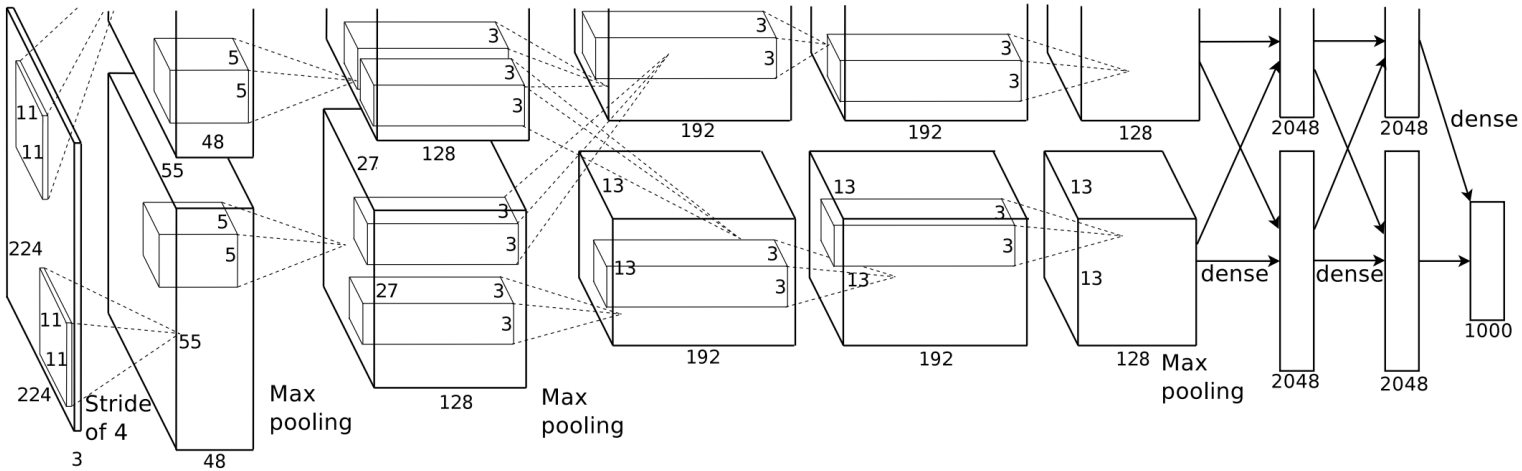


[Kaiming He]



[Canziani et al 2017]

The first ILSVRC winner with deep learning

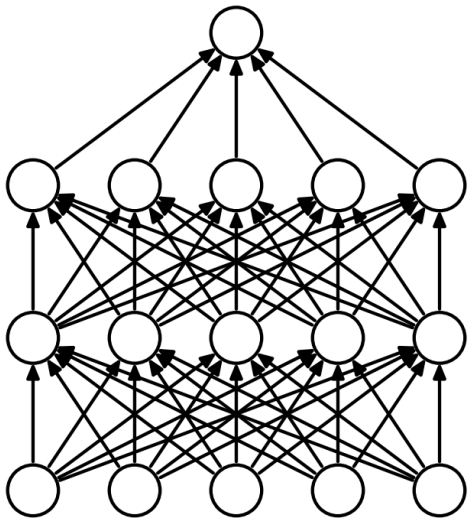


[Krizhevsky et al 2012]

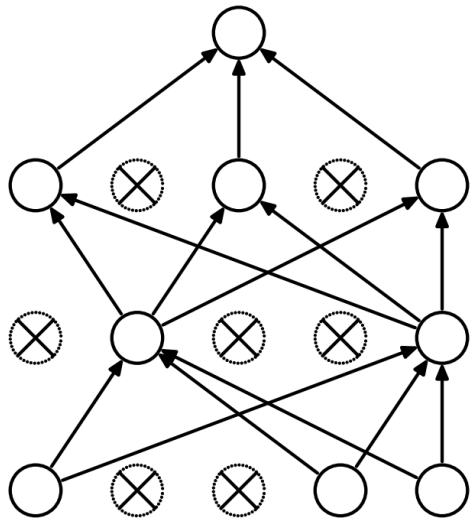
We can understand the entirety of this network

TRICKS OF THE TRADE: DROPOUT

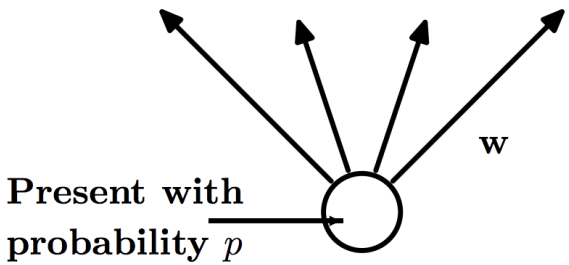
With increasing complexity comes increasing overfitting. Let's regularize!



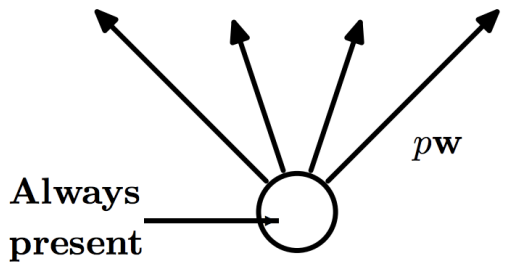
(a) Standard Neural Net



(b) After applying dropout.



(a) At training time



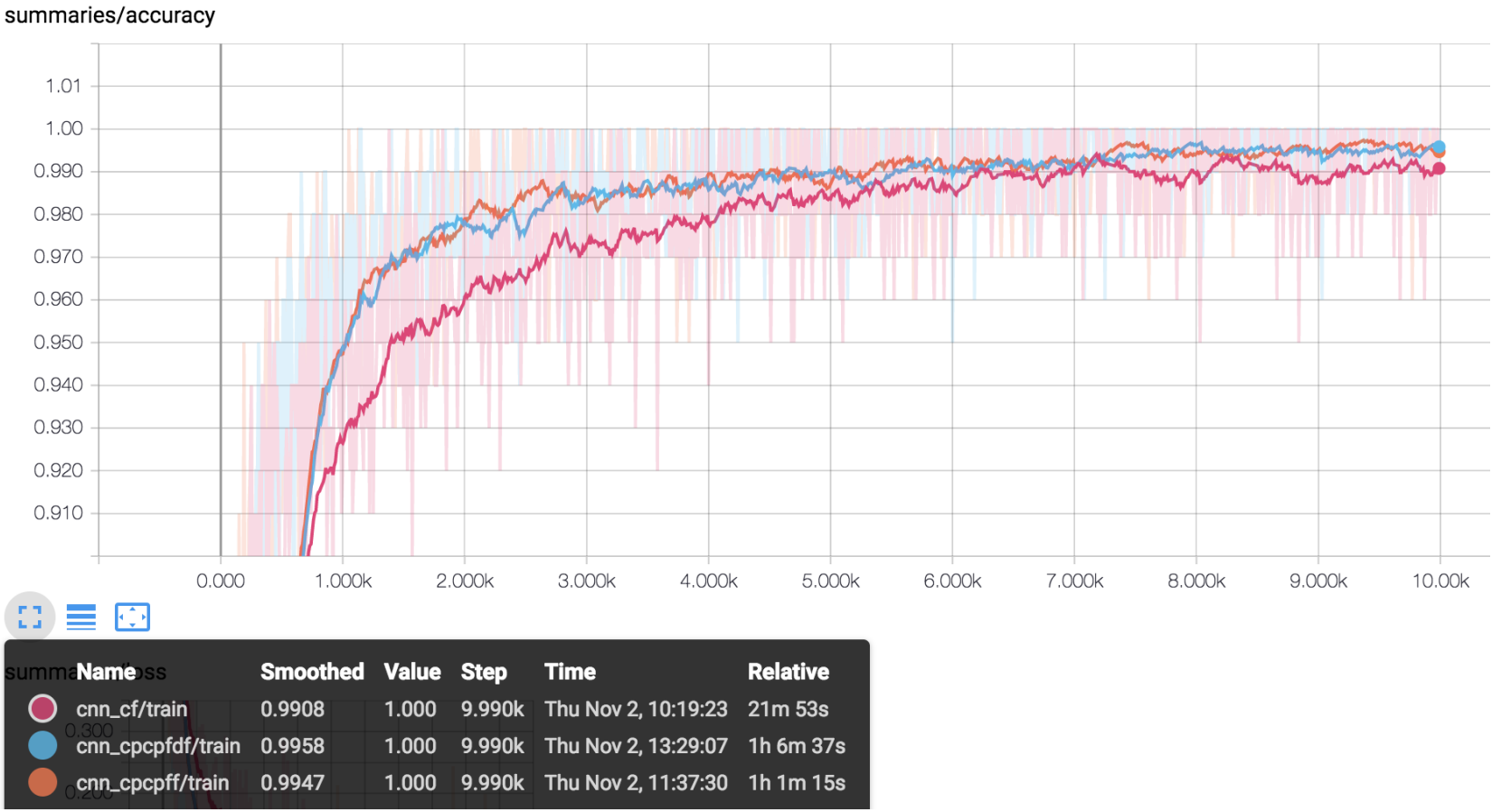
(b) At test time

[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`

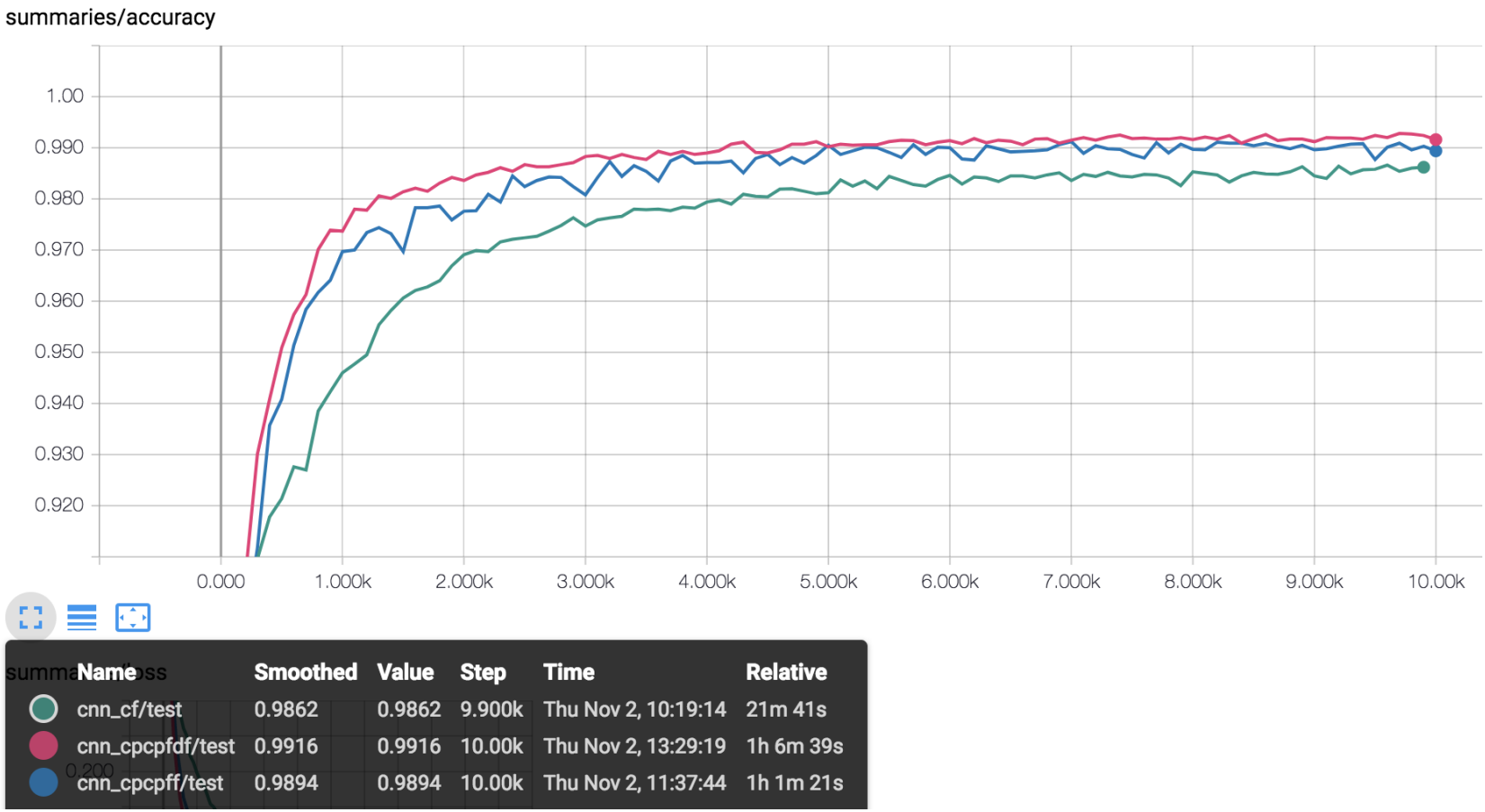


Name	Smoothed	Value	Step	Time	Relative
cnn_cf/train	0.9908	1.000	9.990k	Thu Nov 2, 10:19:23	21m 53s
cnn_cpcpfdf/train	0.9958	1.000	9.990k	Thu Nov 2, 13:29:07	1h 6m 37s
cnn_cpcpff/train	0.9947	1.000	9.990k	Thu Nov 2, 11:37:30	1h 1m 15s

Does not seem to affect training much...

TRICKS OF THE TRADE: DROPOUT

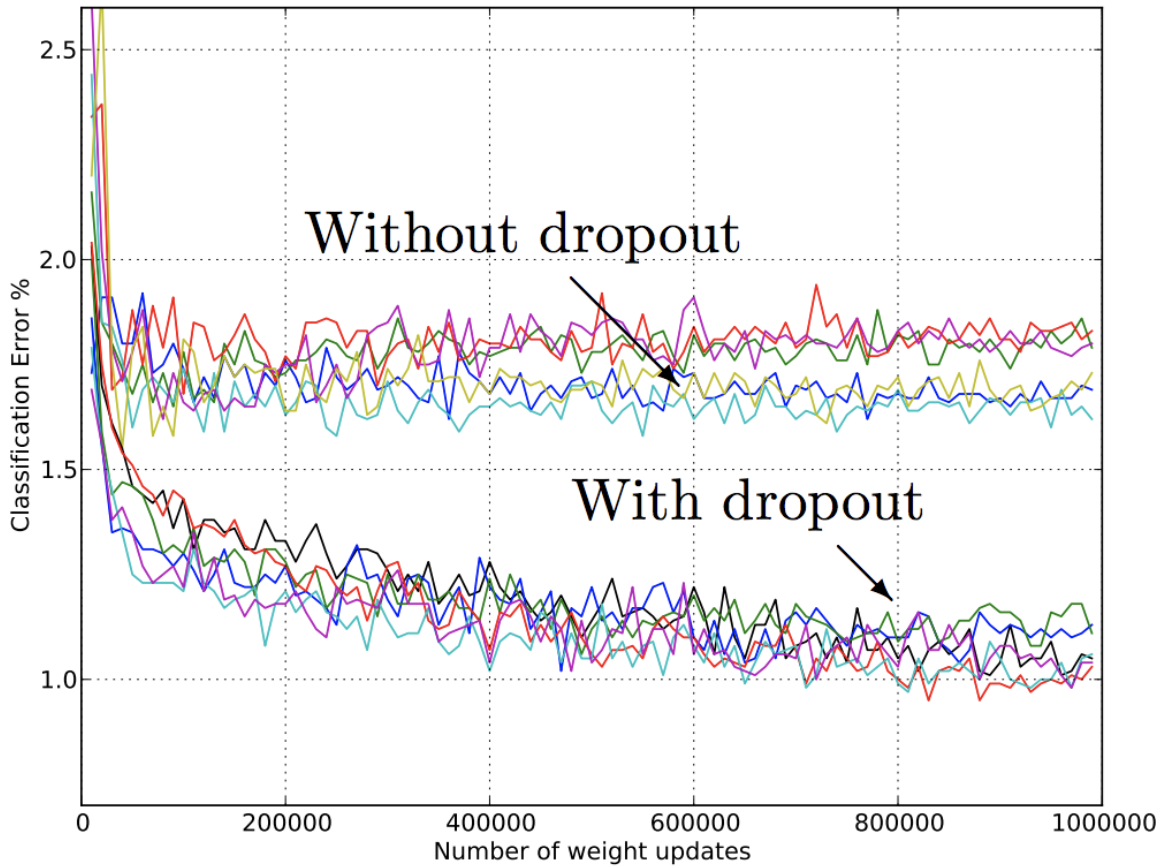
But hopefully it mitigates overfitting



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:

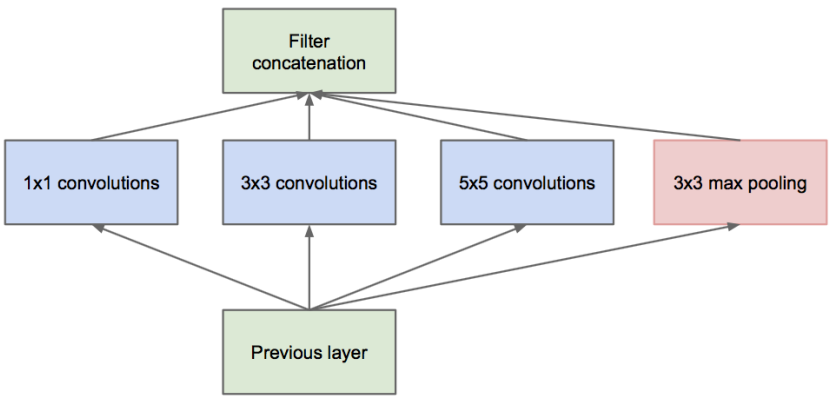
- Change the filters: sizes, striding, padding
- Change the pooling: average/max, different sizes, different positions
- Change the architecture
- Change the optimization method
- Change the batch size
- Change the summary/tensorboard content
- ...

INCEPTION MODULES

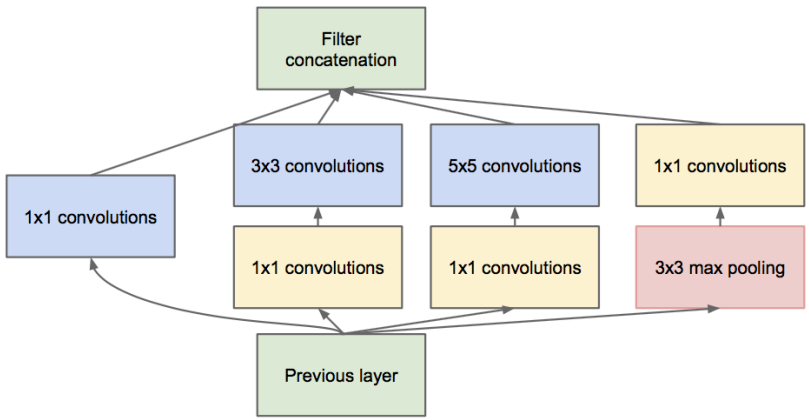
2014 ILSVRC winner added yet more complexity... Idea:

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

Inception module



(a) Inception module, naïve version



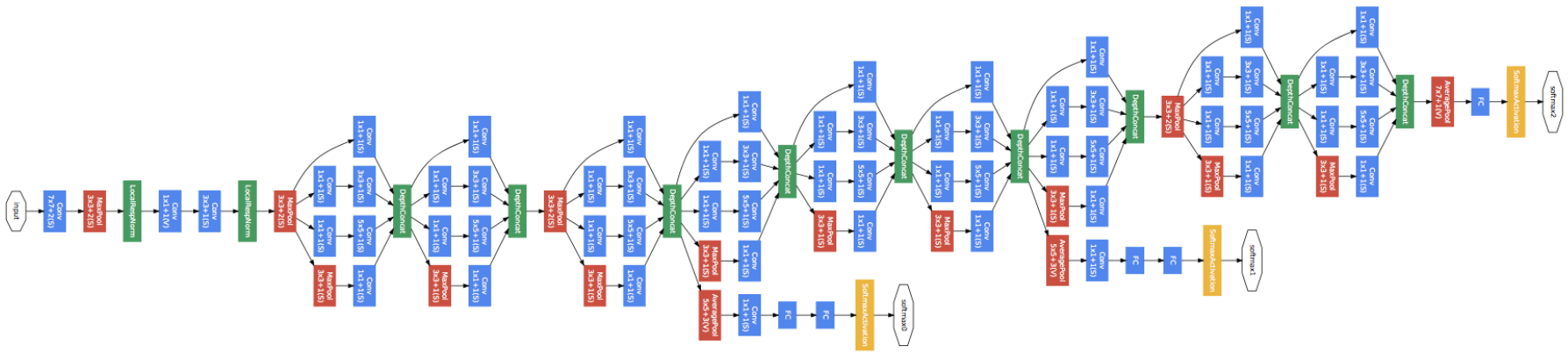
(b) Inception module with dimension reductions

[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

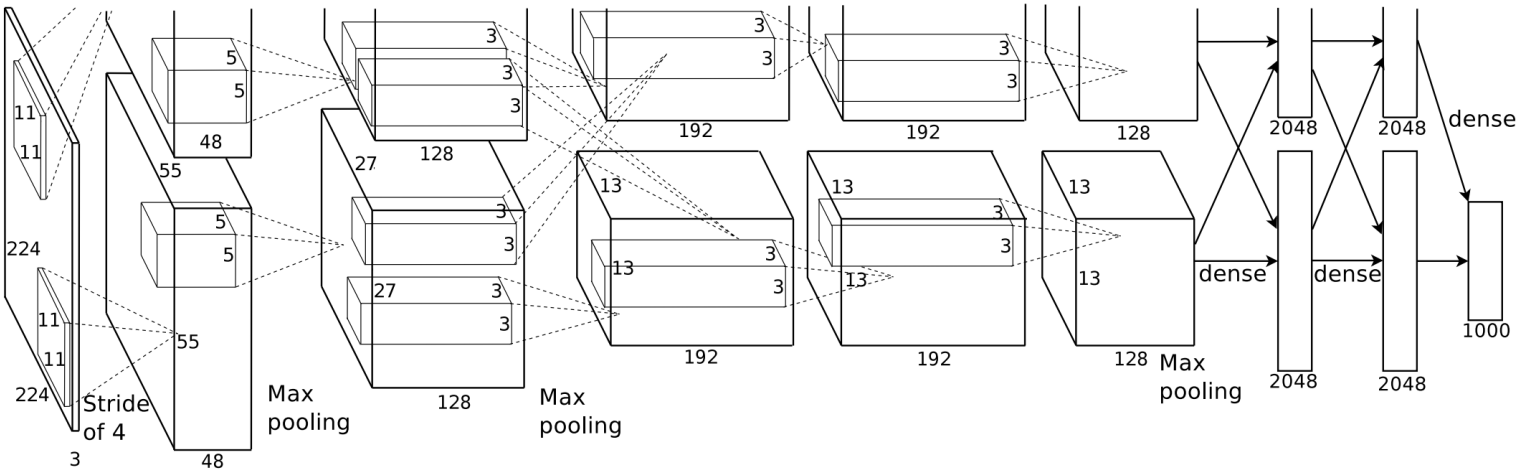
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	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]

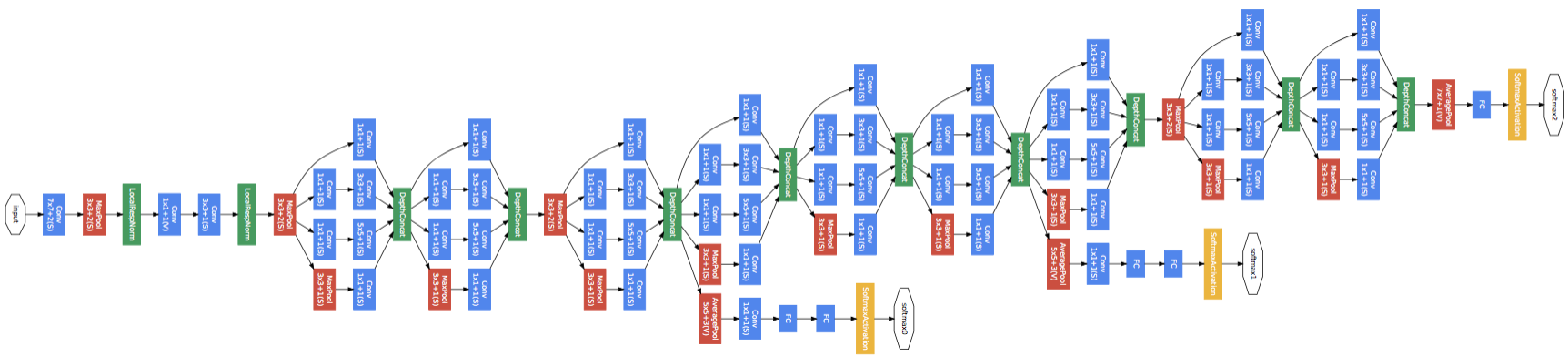
More complex, but still components we understand.

INTERLUDE: RETRAINING / TRANSFER LEARNING

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



[Krizhevsky et al 2012]

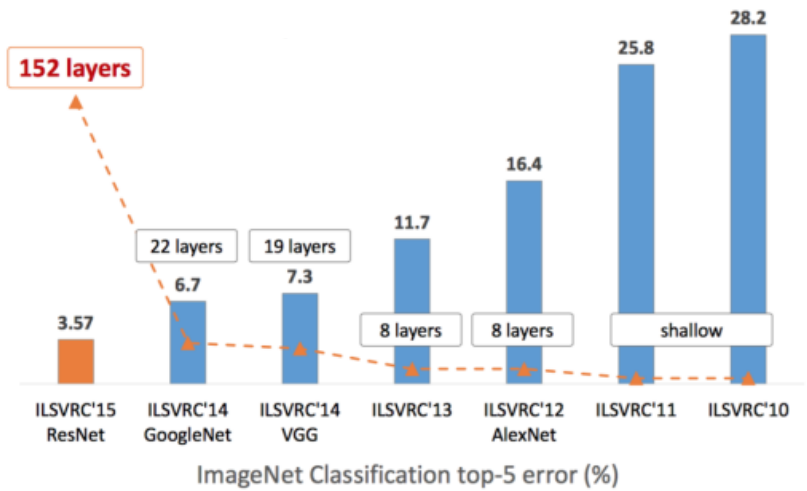


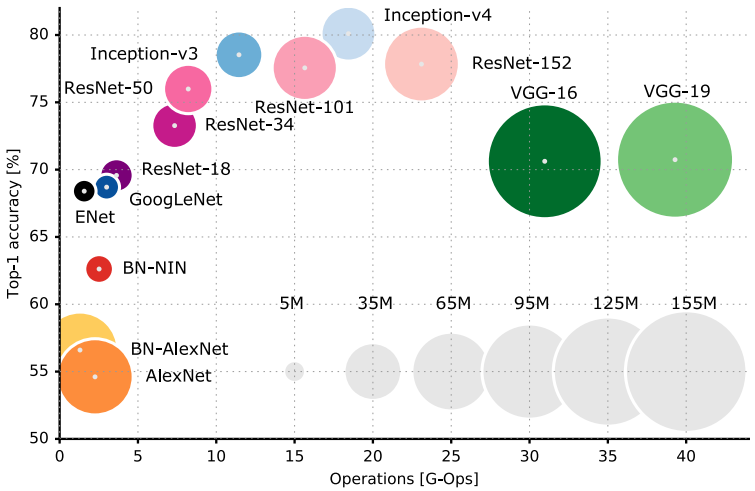
[Szegedy et al 2014]]

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

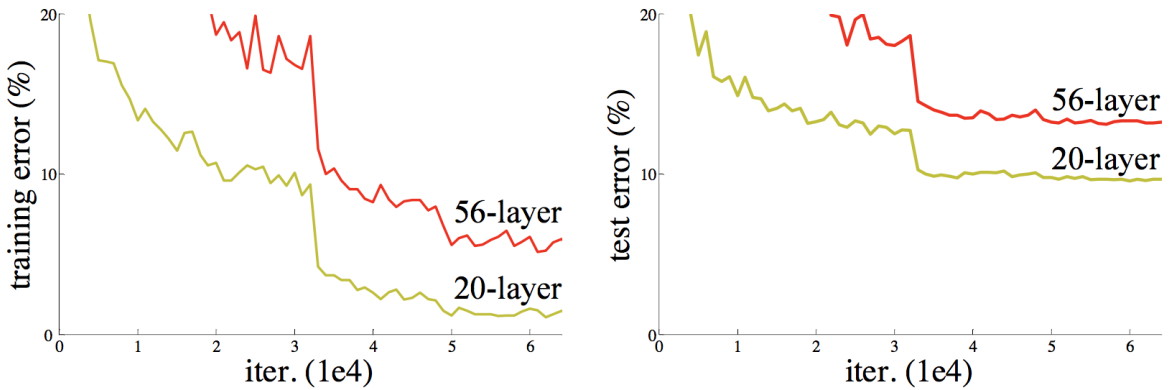
Exploding and *vanishing* gradients were a major historical problem for deep networks

- 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



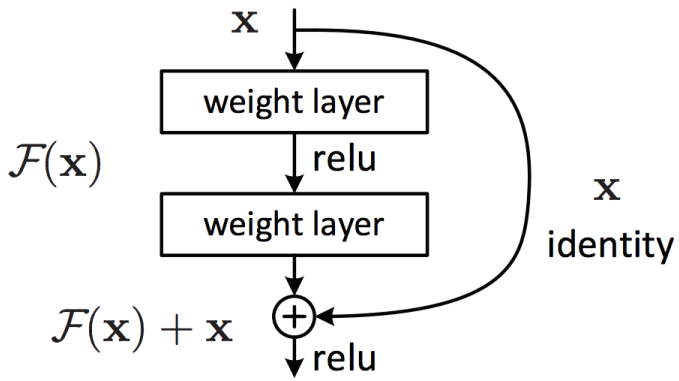
Notice:

- 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...

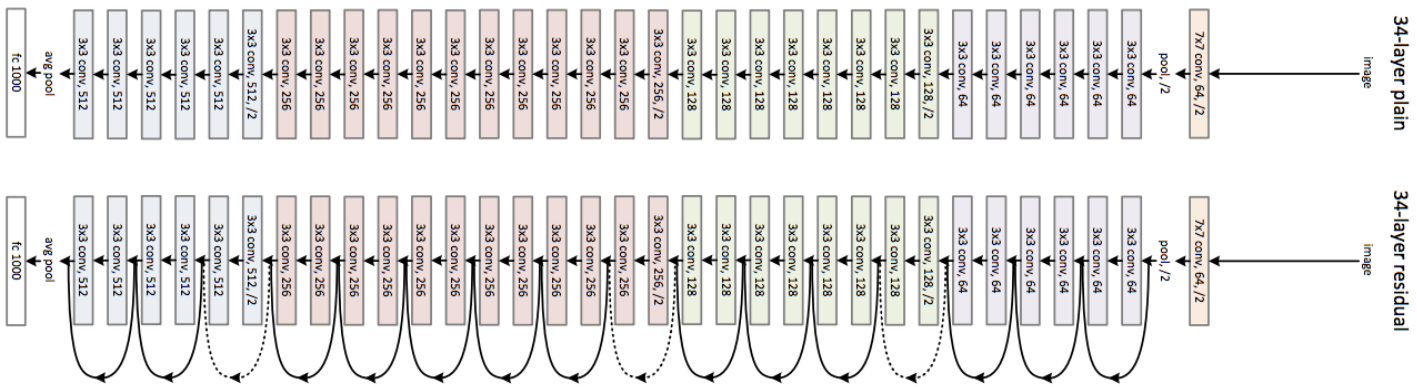
[He et al 2015]

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:

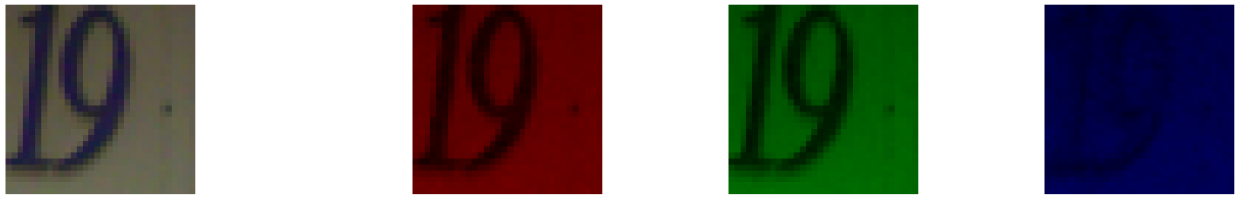
- If $\mathcal{F}$ was well chosen on MNIST, will it work well on SVHN?
- If yes, what does that mean?
- If no, what do we have to change to make it work?
- ...

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

1. DATA

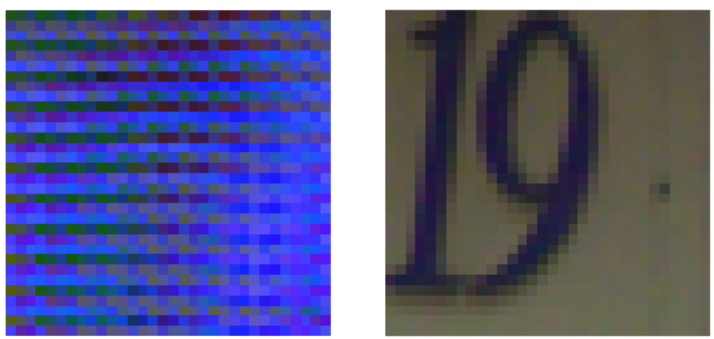
Input layer: three maps of size 32×32

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



- 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
```

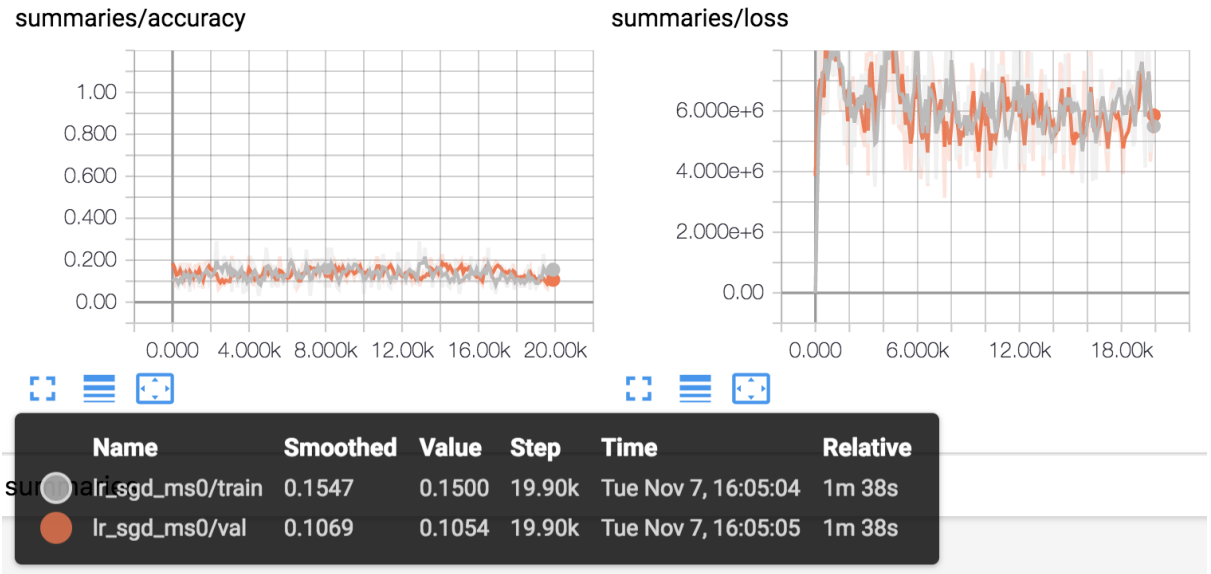
Reminder!

- 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...

3. LOGISTIC REGRESSION AND BASIC DEBUGGING

Start with logistic regression and SGD



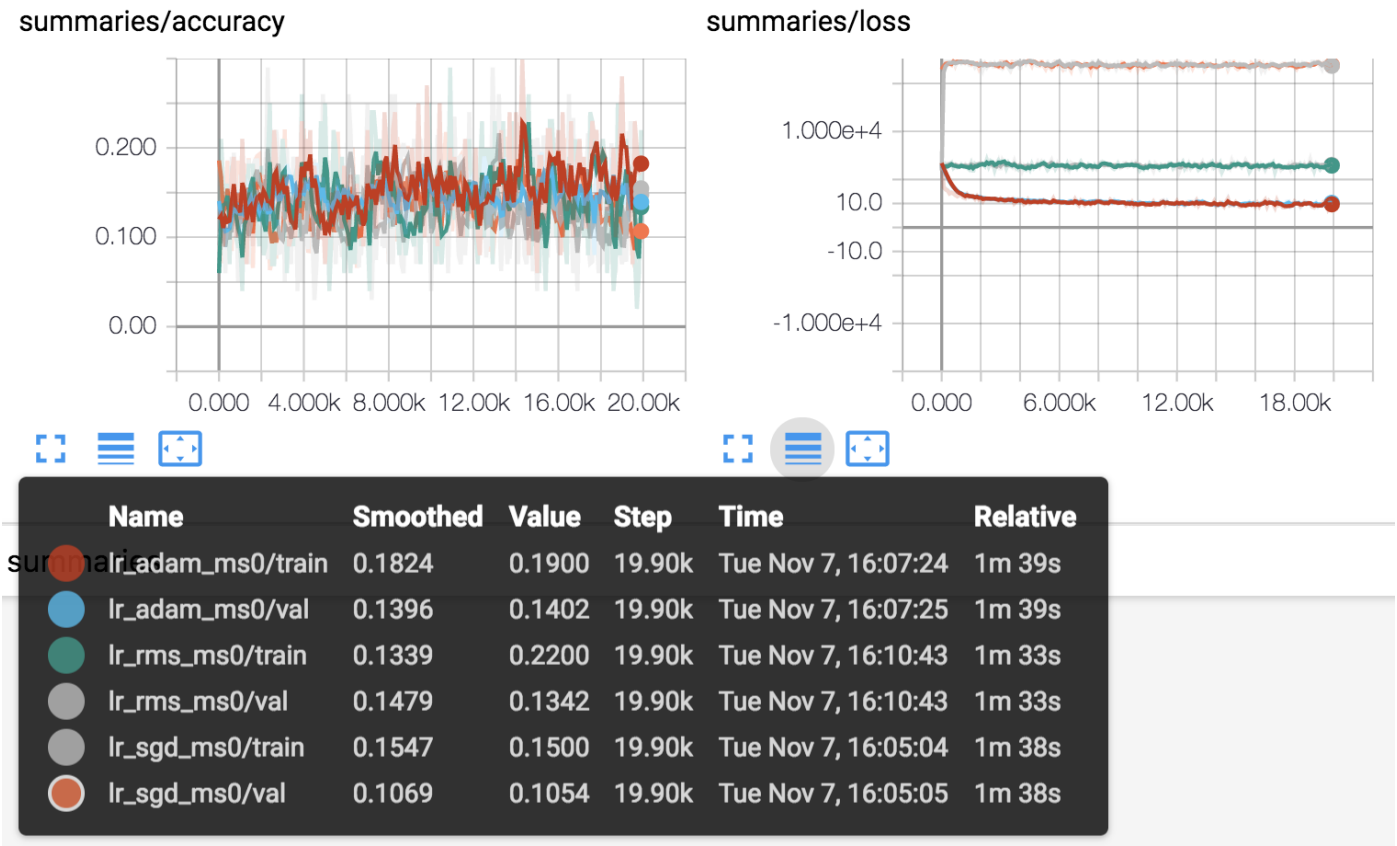
`tb` helps, but basic debugging is still useful

```
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...

4. CHOOSING AN OPTIMIZER

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



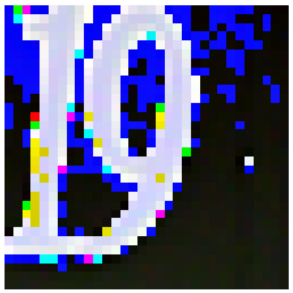
Name	Smoothed	Value	Step	Time	Relative
lr_adam_ms0/train	0.1824	0.1900	19.90k	Tue Nov 7, 16:07:24	1m 39s
lr_adam_ms0/val	0.1396	0.1402	19.90k	Tue Nov 7, 16:07:25	1m 39s
lr_rms_ms0/train	0.1339	0.2200	19.90k	Tue Nov 7, 16:10:43	1m 33s
lr_rms_ms0/val	0.1479	0.1342	19.90k	Tue Nov 7, 16:10:43	1m 33s
lr_sgd_ms0/train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

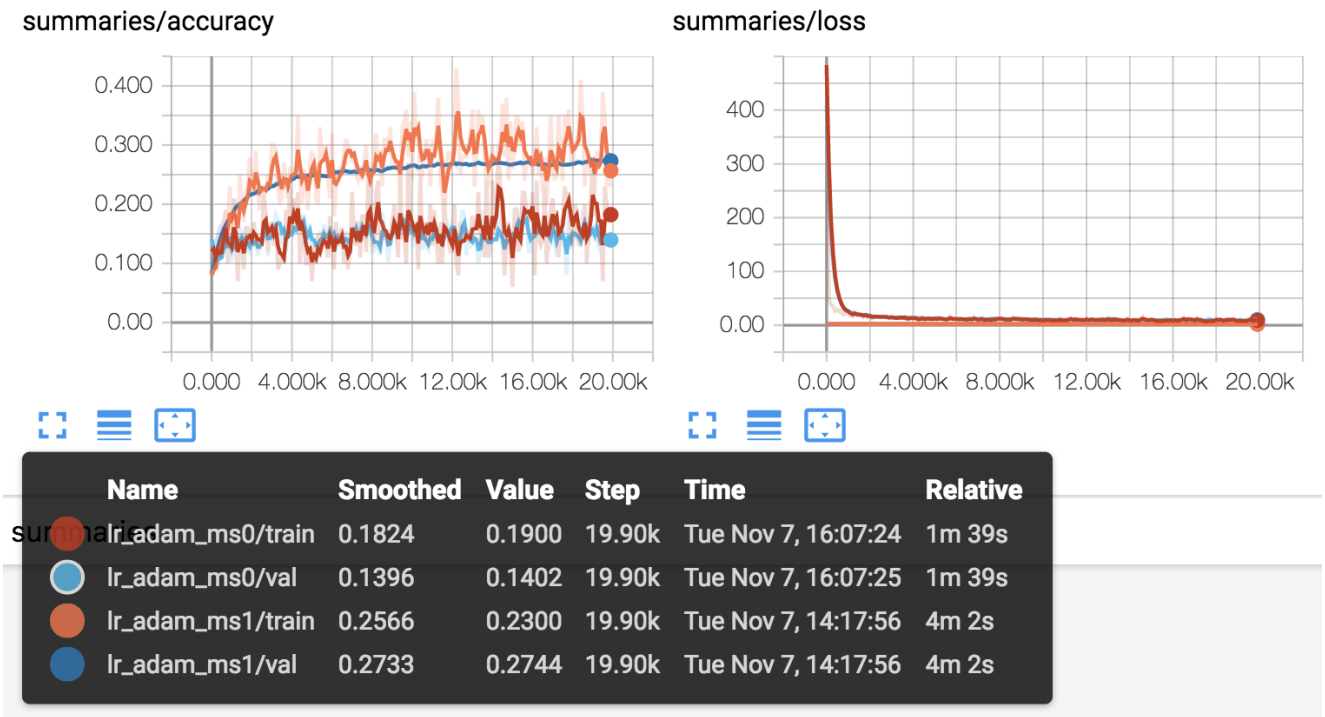
Performance is still terrible, but at least the loss function is not pathological. Progress...

5. MEAN SUBTRACTION

Observation

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

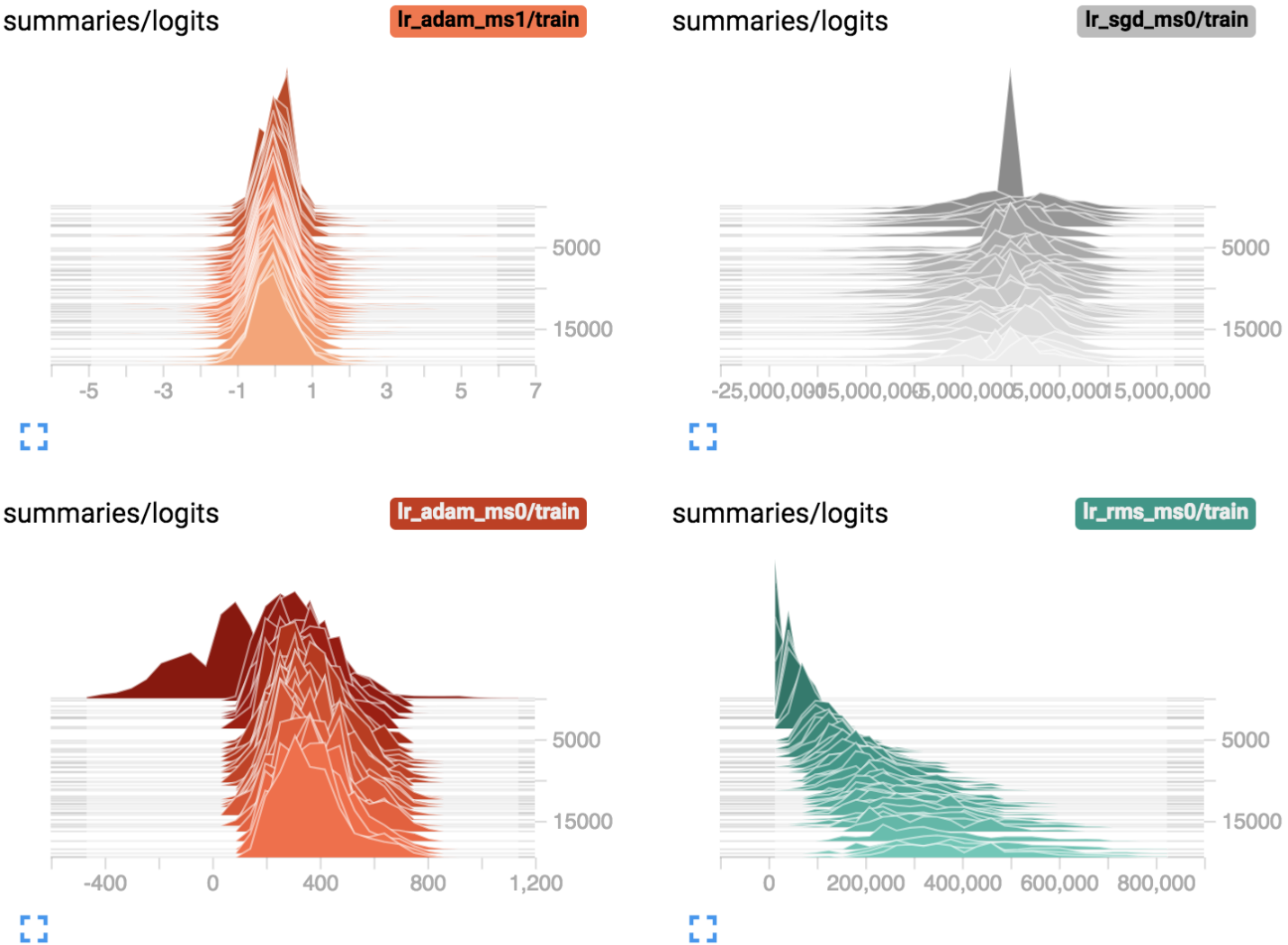




Progress! Preprocessing data matters... do not rely on the neural net to do all the work

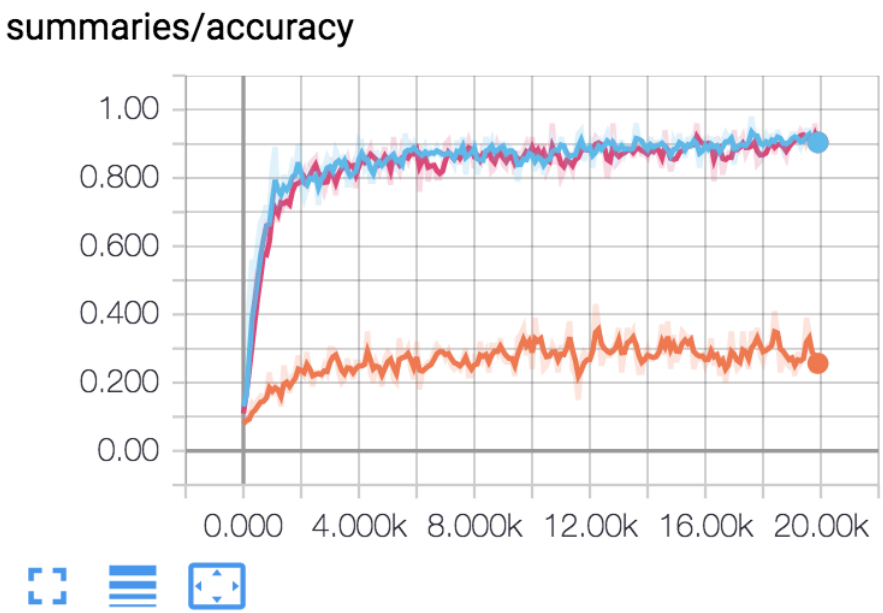
6. TENSORBOARD FOR EMPIRICISM

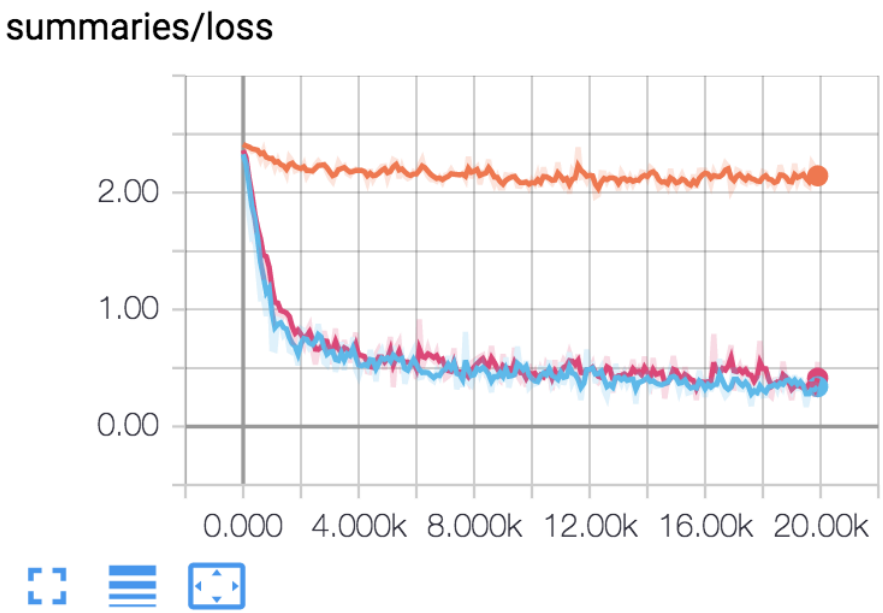
Look at the histograms of logits over time to choose which one is learning.



7. ADDING COMPLEXITY

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

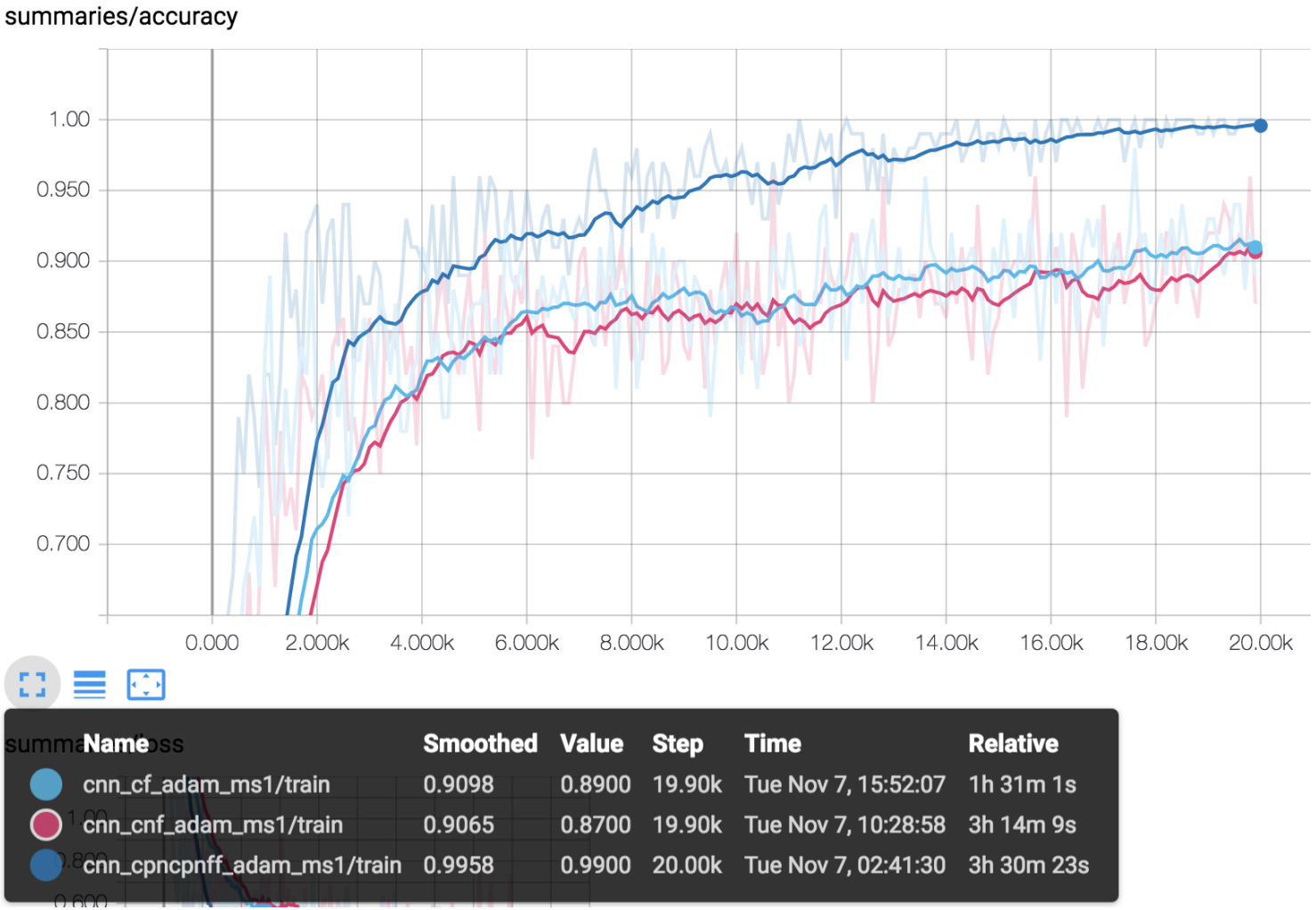




Name	Smoothed	Value	Step	Time	Relative
cnn_cf_adam_ms1/train	0.9040	0.8900	19.90k	Tue Nov 7, 15:52:07	1h 31m 1s
cnn_cnf_adam_ms1/train	0.9048	0.8700	19.90k	Tue Nov 7, 10:28:58	3h 14m 9s
lr_adam_ms1/train	0.2566	0.2300	19.90k	Tue Nov 7, 14:17:56	4m 2s

7. ADDING COMPLEXITY

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



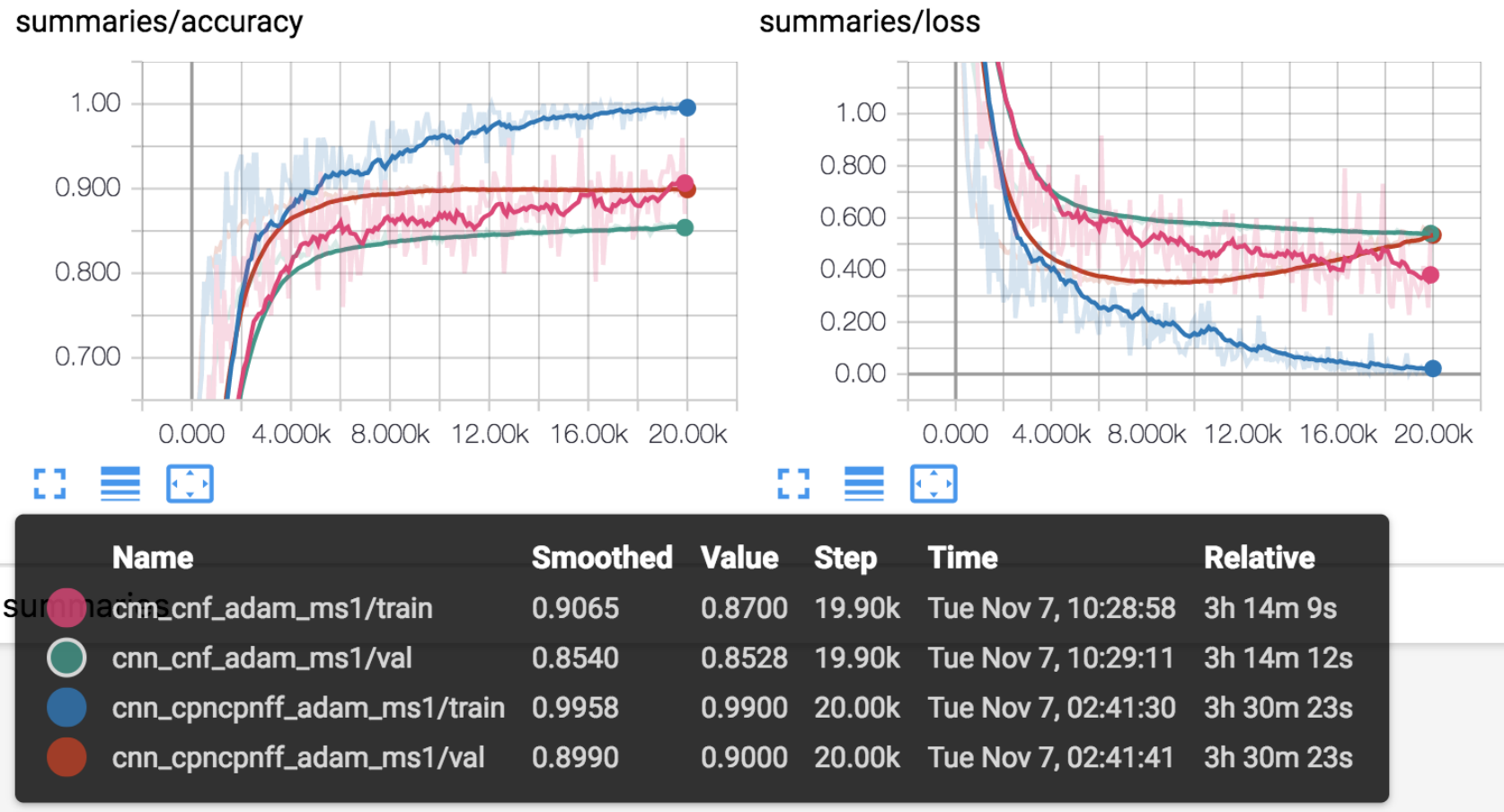
Name	Smoothed	Value	Step	Time	Relative
cnn_cf_adam_ms1/train	0.9098	0.8900	19.90k	Tue Nov 7, 15:52:07	1h 31m 1s
cnn_cnf_adam_ms1/train	0.9065	0.8700	19.90k	Tue Nov 7, 10:28:58	3h 14m 9s
cnn_cpncpnff_adam_ms1/train	0.9958	0.9900	20.00k	Tue Nov 7, 02:41:30	3h 30m 23s

Training performance is very high. Overfitting?

8. VALIDATION DATA

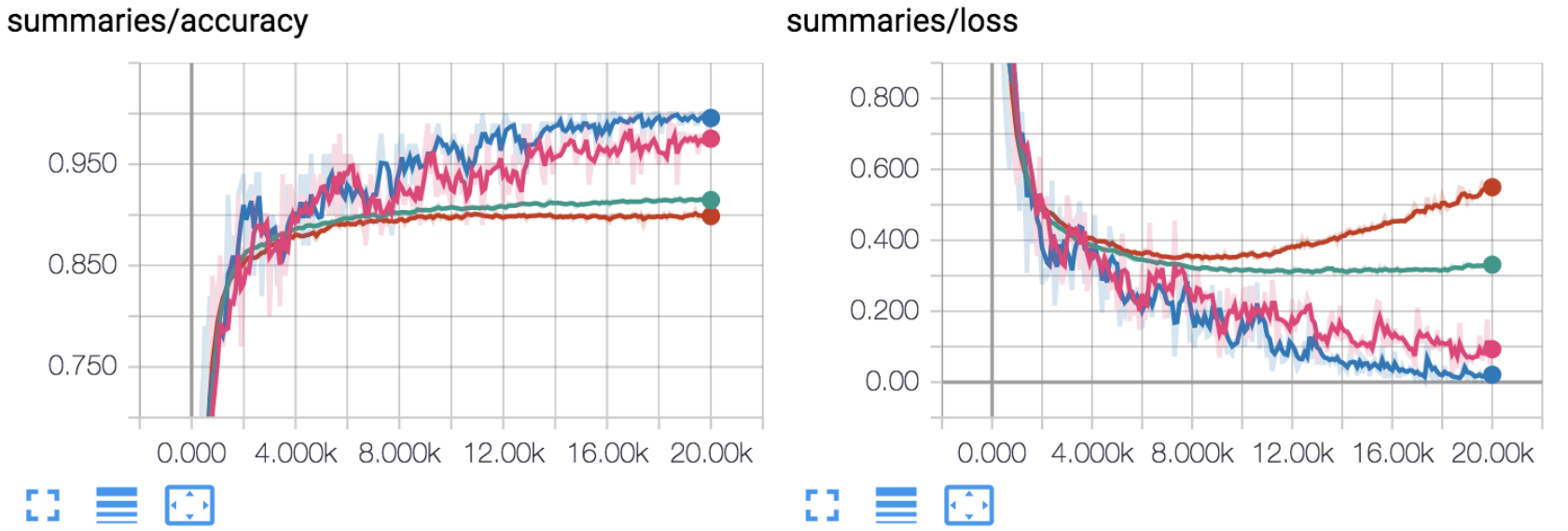
A separate validation set:

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



9. DROPOUT

Add a dropout layer to regularize



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.

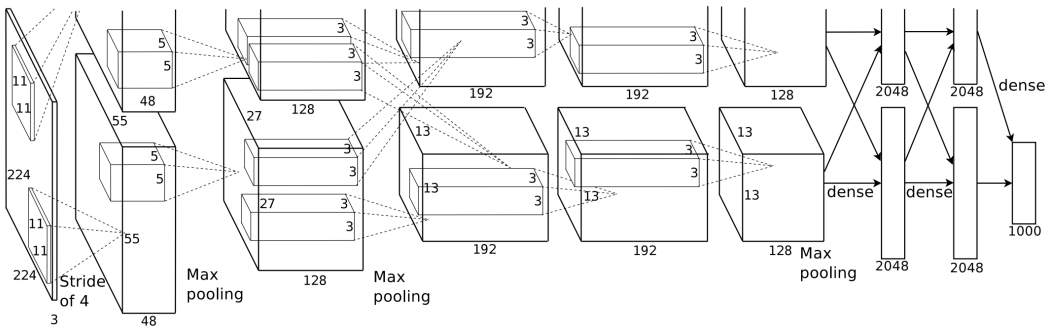
Tools and tricks at your disposal:

- 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.?