STAT GR5242: Advanced Machine Learning
Lecture slides: Weeks 1-3

John Cunningham

Department of Statistics
Columbia University

Administrative reminders

Welcome! Let’s discuss the syllabus...

CONTEXT: 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 $\mathcal{F} = \{f_\theta : \theta \in \Theta\}$

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

- Specify a loss function between model and data

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

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

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

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

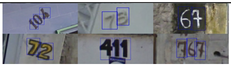
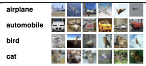
The four catalysts of the AI explosion

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

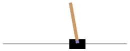
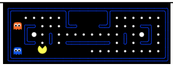
We will study some neural network families $\mathcal{F}$. While neural networks are powerful, there is nothing magical or fundamentally different than what you already know.

Catalyst 1: Data

Computer Vision

SVHN	CIFAR10	ImageNet	...
			...

Reinforcement Learning

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

Natural Language Processing

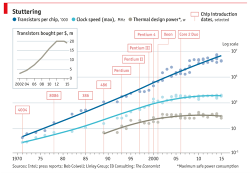
Wikipedia (English)	Twitter	Jeopardy	...
			...

And so much more...

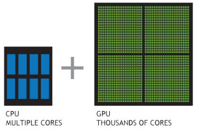
- `https://www.data.gov/`
- `https://opendata.cityofnewyork.us/`
- `https://github.com/caesar0301/awesome-public-datasets`
- ...

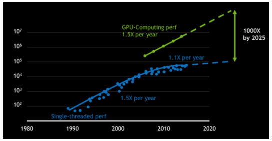
Catalyst 2: computational power

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



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



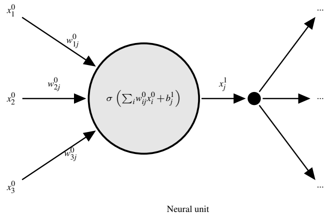


Cloud computing has made this even easier (abstracting away IT and system ops)

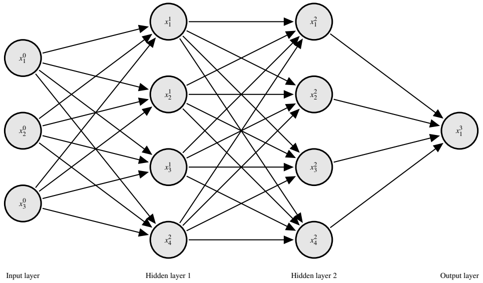
amazon web services™

Google Cloud Platform

CATALYST 3: NEURAL NETWORKS



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...).

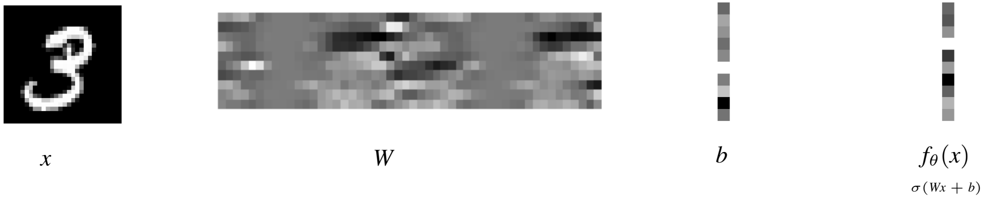
- Inputs x_i^0 enter into unit j, weighted by edges w_{ij}^0, and are summed with bias b_j^1
- $\sigma(\cdot)$ provides elementwise nonlinearity
- The result x_j^1 is transmitted to layer 2, the next layer

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

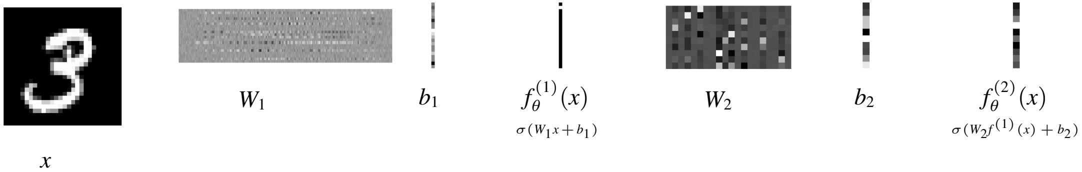
$$\theta = \left\{ w_{ij}^\ell, b_j^\ell \right\}_{i,j,\ell} = \{W_\ell, b_\ell\}_\ell$$

Example: logistic regression → neural networks

Logistic Regression

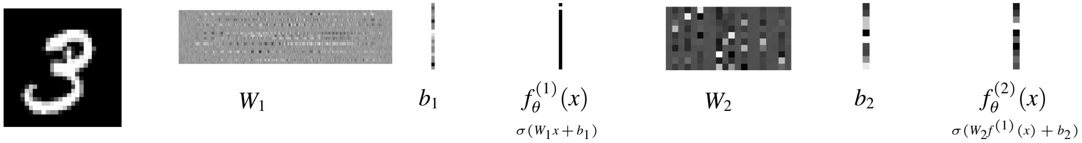


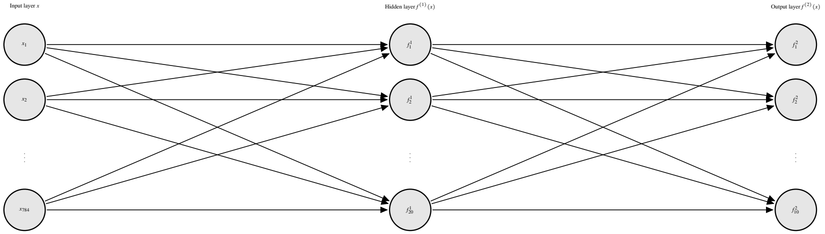
Neural Network



EXAMPLE: LOGISTIC REGRESSION → NEURAL NETWORKS

Neural Network





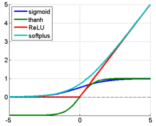
Cascade layers for any amount of depth and complexity!

Naive conclusion: deep learning is easy...

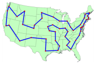
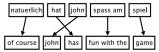
...Deep learning is hard

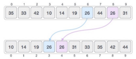
- How do I choose $\left|f^{(1)}\right|$, the number of *units* in the hidden layers?
- How do I choose L, the number of *layers*?
- How do I choose the *activation function* $\sigma(\cdot)$?

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

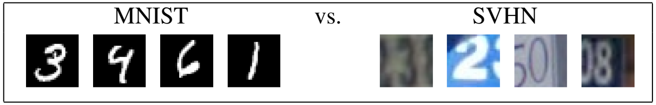


- Are there other choices to make?
- What about overfitting?
- Will my optimizer converge?
- Is my problem solvable with a particular *architecture* $\mathcal{F}$?



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



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

Catalyst 4: software

Machine Learning libraries have abstracted $\{$math, stats, optimization, ...$\} \rightarrow$ engineering

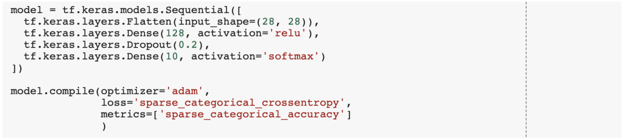
TensorFlow™ K Keras torch Caffe ...

Under the hood are several essential elements to understand:

- Neural networks in detail

(sounds obvious, but we'll spend some time here...)

- 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

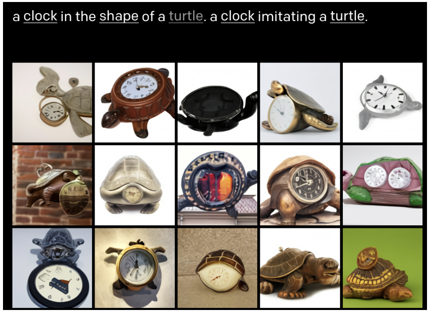
(much more to come here also...)

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

Neural Networks

Administrative reminders

- Slides and syllabus on courseworks (and Assignment 1 soon)
- A few comments about textbooks:
  - There is no textbook for this course... for a good reason.
  - When there is a relevant background reading or survey/review, I will note it in class.
  - *Mathematics for Machine Learning* A. Aldo Faisal, Cheng Soon Ong, and Marc Peter Deisenroth
  - *Probabilistic Machine Learning* Kevin P. Murphy
  - *Deep Learning* Aaron Courville, Yoshua Bengio, Ian Goodfellow
  - *Pattern Recognition and Machine Learning* Christopher Bishop
- Ask questions in class. Don't wait until after class and then divide the impact of that question by 100x.
- Also, so you don't think I'm just making stuff up, a DALL-E sample:


a clock in the shape of a turtle. a clock imitating a turtle.

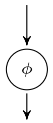
A Most Important Reminder

A neural network represents a function $f_\theta : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$.

BUILDING BLOCKS

Units

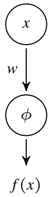
The basic building block is a **node** or **unit**:



- The unit has incoming and outgoing arrows. We think of each arrow as "transmitting" a signal.
- The signal is always a scalar.
- A unit represents a function ϕ.

We read the diagram as: A scalar value (say x) is transmitted to the unit, the function ϕ is applied, and the result $\phi(x)$ is transmitted from the unit along the outgoing arrow.

Weights

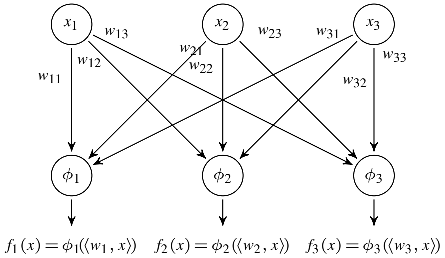


- If we want to "input" a scalar x, we represent it as a unit, too.
- We can think of this as the unit representing the constant function $g(x) = x$.
- Additionally, each arrow is usually inscribed with a (scalar) weight w.
- As the signal x passes along the edge, it is multiplied by the edge weight w.

The diagram above represents the function $f(x) := \phi(wx)$.

Reading Neural Networks

$$f : \mathbb{R}^3 \to \mathbb{R}^3 \quad \text{with input} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$



x_1
x_2
x_3
w_{11}
w_{12}
w_{13}
w_{21}
w_{22}
w_{23}
w_{31}
w_{32}
w_{33}
ϕ_1
ϕ_2
ϕ_3
$f_1(x) = \phi_1(\langle w_1, x\rangle)$
$f_2(x) = \phi_2(\langle w_2, x\rangle)$
$f_3(x) = \phi_3(\langle w_3, x\rangle)$

$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ f_3(x) \end{pmatrix} \quad \text{with} \quad f_i(x) = \phi_i\Big(\sum_{j=1}^{3} w_{ji} x_j\Big)$$

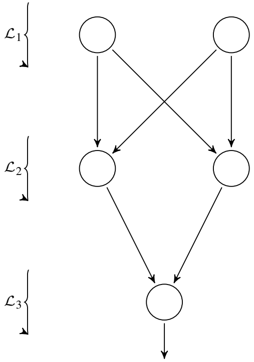
(recall inner product $\langle w_i, x\rangle = w_i^\top x = \sum_j w_{ji} x_j$)

Feed-Forward Networks

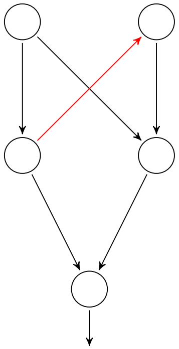
A **feed-forward network** is a neural network whose units can be arranged into groups $\mathcal{L}_1, \dots, \mathcal{L}_K$ so that connections (arrows) only pass from units in group $\mathcal{L}_k$ to units in group $\mathcal{L}_{k+1}$. The groups are called **layers**. In a feed-forward network:

- There are no connections within a layer.
- There are no backwards connections.
- There are no connections that skip layers, e.g. from $\mathcal{L}_k$ to units in group $\mathcal{L}_{k+2}$.

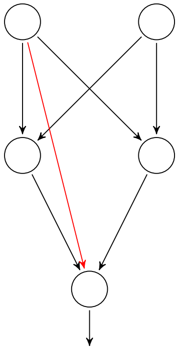
(but see Huang...Weinberger 2017 CVPR)



feed-forward

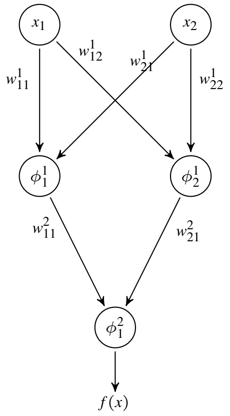


not feed-forward



not feed-forward (but still useful...)

Layers



- This network computes the function

$$f(x_1,x_2) = \phi^2_1\Big(w^2_{11}\phi^1_1\big(w^1_{11}x_1 + w^1_{21}x_2\big) + w^2_{21}\phi^1_2\big(w^1_{12}x_1 + w^1_{22}x_2\big)\Big)$$

- Clearly, writing out f gets complicated fairly quickly as the network grows.

First shorthand: Scalar products

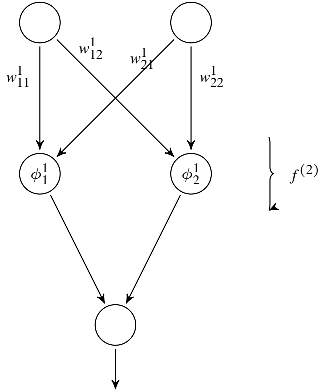
- Collect all weights coming into a unit into a vector, e.g.

$$w^2_1 := (w^2_{11}, w^2_{21})$$

- Same for inputs: $x = (x_1, x_2)$
- The function then becomes

$$f(x) = \phi^2_1\left(\left\langle w^2_1, \begin{pmatrix}\phi^1_1(\langle w^1_1, x\rangle)\\ \phi^1_2(\langle w^1_2, x\rangle)\end{pmatrix}\right\rangle\right)$$

Layers



- Each layer represents a function, which takes the output values of the previous layers as its arguments.
- Suppose the output values of the two nodes at the top are y_1, y_2.
- Then the second layer defines the (two-dimensional) function

$$f^{(2)}(y) = \begin{pmatrix} \phi^1_1(\langle w^1_1, y\rangle) \\ \phi^1_2(\langle w^1_2, y\rangle) \end{pmatrix}$$

Composition of Functions

Basic composition

Suppose f and g are two function $\mathbb{R} \to \mathbb{R}$. Their **composition** $g \circ f$ is the function

$$g \circ f(x) := g(f(x)) \ .$$

For example:

$$f(x) = x + 1 \qquad g(y) = y^2 \qquad g \circ f(x) = (x+1)^2$$

We could combine the same functions the other way around:

$$f \circ g(x) = x^2 + 1$$

In multiple dimensions

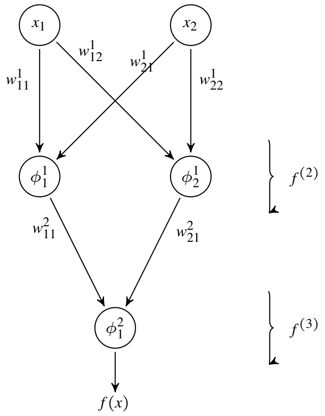
Suppose $f : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ and $g : \mathbb{R}^{d_2} \to \mathbb{R}^{d_3}$. Then

$$g \circ f(x) = g(f(x)) \qquad \text{is a function } \mathbb{R}^{d_1} \to \mathbb{R}^{d_3} \ .$$

For example:

$$f(x) = \langle x, v \rangle - c \qquad g(y) = \operatorname{sgn}(y) \qquad g \circ f(x) = \operatorname{sgn}(\langle x, v \rangle - c)$$

LAYERS AND COMPOSITION



- As above, we write

$$f^{(2)}(\,\bullet\,) = \begin{pmatrix} \phi^1_1(\langle w^1_1, \bullet \rangle) \\ \phi^1_2(\langle w^1_2, \bullet \rangle) \end{pmatrix}$$

- The function for the third layer is similarly

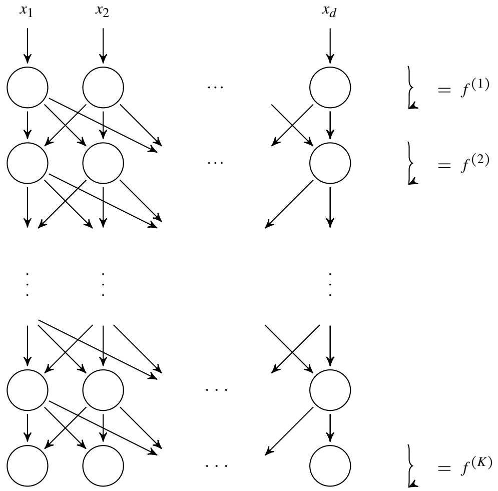
$$f^{(3)}(\,\bullet\,) = \phi^2_1(\left\langle w^2_1, \bullet \right\rangle)$$

- The entire network represents the function

$$f(x) = f^{(3)}(f^{(2)}(x)) = f^{(3)} \circ f^{(2)}(x)$$

A feed-forward network represents a function as a composition of several functions, each given by one layer.

THE FULLY CONNECTED DEEP NETWORK (OFTEN "MLP")



$$f(x) \quad = \quad f^{(K)}(\cdots f^{(2)}(f^{(1)}(x))) \quad = \quad f^{(K)} \circ \ldots \circ f^{(1)}(x)$$

Layers and Compositions

General feed-forward networks

A feed-forward network with K layers represents a function

$$f(x) \quad = \quad f^{(K)} \circ \ldots \circ f^{(1)}(x)$$

Each layer represents a function $f^{(k)}$. These functions are of the form:

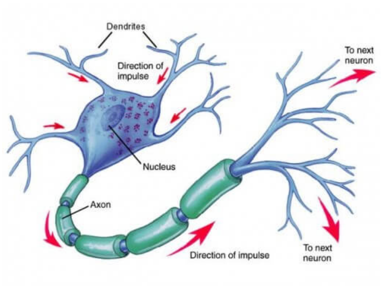
$$f^{(k)}(\,\bullet\,) = \begin{pmatrix} \phi_1^{(k)}(\langle w_1^{(k)}, \bullet \rangle) \\ \vdots \\ \phi_d^{(k)}(\langle w_d^{(k)}, \bullet \rangle) \end{pmatrix} \qquad \text{typically:} \qquad \phi^{(k)}(x) = \begin{cases} \sigma(x) & \text{(sigmoid)} \\ \mathbb{I}\{\pm x > \tau\} & \text{(threshold)} \\ c & \text{(constant)} \\ x & \text{(linear)} \\ \max\{0, x\} & \text{(rectified linear)} \end{cases}$$

Dimensions

- Each function $f^{(k)}$ is of the form
$$f^{(k)} : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$$
- d_k is the number of nodes in the kth layer. It is also called the *width* of the layer.
- We mostly assume for simplicity: $d_1 = \ldots = d_K =: d$.

ORIGIN OF THE NAME

If you look up the term “neuron” online, you will find illustrations like this:



This one comes from a web site called easyscienceforkids.com, which means it is likely to be scientifically more accurate than typical references to “neuron” and “neural” in machine learning.

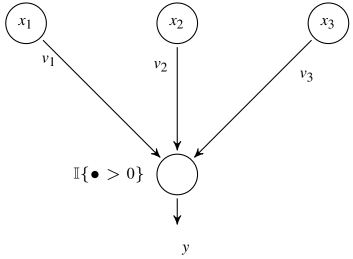
Very roughly, a neuron is a cell that:

- Collects signals (often electrical, often from other neurons)
- Processes them
- Generates an output signal

What happens inside a neuron is an intensely studied problem in neuroscience and is far more complex than this three-step concept, so only in the rarest settings is there any connection between deep learning and “understanding the brain”.

Historical perspective: McCulloch-Pitts Neuron

A neuron is modeled as a "thresholding device" that combines input signals:

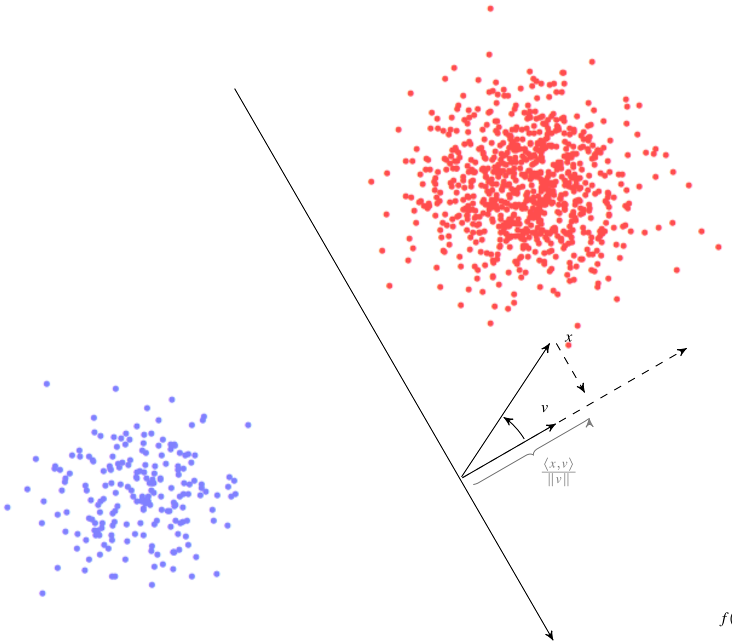


McCulloch-Pitts neuron model (1943)

- Collect the input signals x_1, x_2, x_3 into a vector $x = (x_1, x_2, x_3) \in \mathbb{R}^3$
- Choose fixed vector $v \in \mathbb{R}^3$ and constant $c \in \mathbb{R}$.
- Compute:

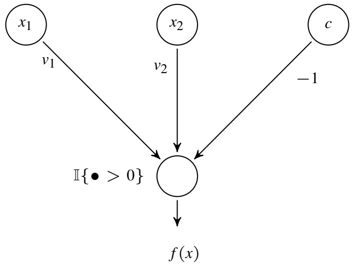
$$y = \mathbb{I}\{\langle v, x\rangle > 0\} \qquad \text{for some } c \in \mathbb{R}\ .$$

- In hindsight, this is a neural network with two layers, and function $\phi(\,\bullet\,) = \mathbb{I}\{\langle v, x\rangle > 0\}$ at the bottom unit.



$$f(x) = \operatorname{sgn}(\langle v, x\rangle - c)$$

LINEAR CLASSIFIER IN $\mathbb{R}^2$ AS TWO-LAYER NN



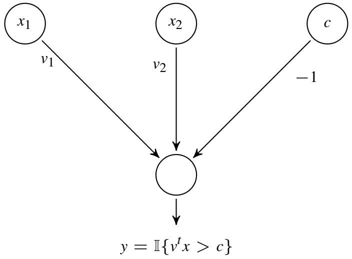
$$f(x) \quad = \quad \mathbb{I}\{\, v_1x_1 + v_2x_2 + (-1)c \;>\; 0 \,\} \quad = \quad \mathbb{I}\{\langle \mathbf{v}, \mathbf{x}\rangle > c\}$$

Equivalent to linear classifier

The linear classifier on the previous slide and f differ only in whether they encode the “blue” class as -1 or as 0:

$$\operatorname{sgn}(\langle v, x\rangle - c) \;=\; 2f(x) - 1$$

Remarks

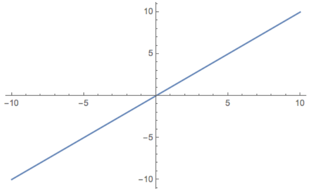


- This neural network represents a linear two-class classifier (on $\mathbb{R}^2$).
- We can more generally define a classifier on $\mathbb{R}^d$ by adding input units, one per dimension.
- It does not specify the training method.
- To train the classifier, we need a loss function (for ERM!) and an optimization method.

Typical Component Functions

Linear units

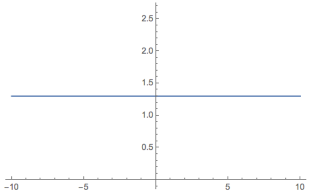
$$\phi(x) = x$$



This function simply "passes on" its incoming signal. These are used for example to represent inputs (data values).

Constant functions

$$\phi(x) = c$$

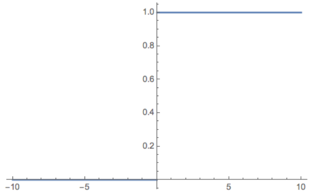


These can be used e.g. in combination with an indicator function to define a threshold, as in the linear classifier above.

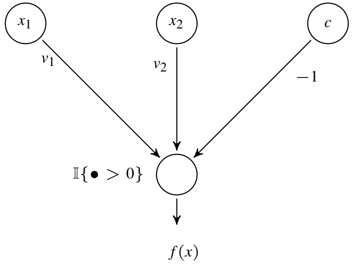
Typical Component Functions

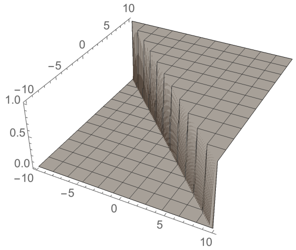
Indicator function

$$\phi(x) = \mathbb{I}\{x > 0\}$$



Example: Final unit is indicator

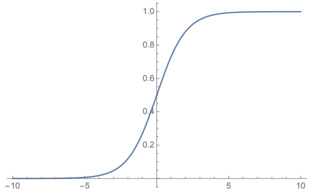




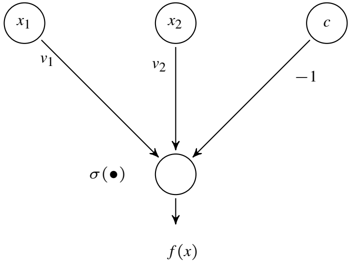
Typical Component Functions

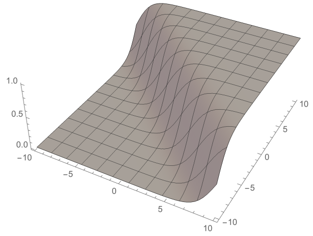
Sigmoids

$$\phi(x) = \frac{1}{1 + e^{-x}}$$



Example: Final unit is sigmoid

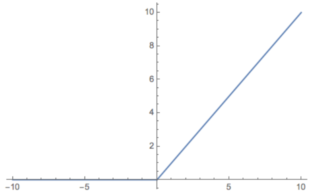




Typical Component Functions

Rectified linear units

$$\phi(x) = \max\{0, x\}$$



These are currently the most commonly used unit in the "inner" layers of a neural network (those layers that are not the input or output layer).

HIDDEN LAYERS AND NONLINEAR FUNCTIONS

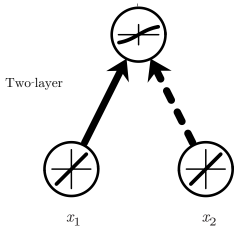
Hidden units

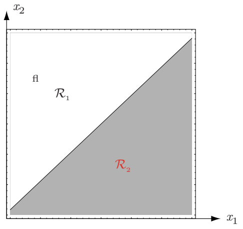
- Any nodes (or “units”) in the network that are neither input nor output nodes are called **hidden**.
- Every network has an input layer and an output layer.
- If there any additional layers (which hence consist of hidden units), they are called **hidden layers**.

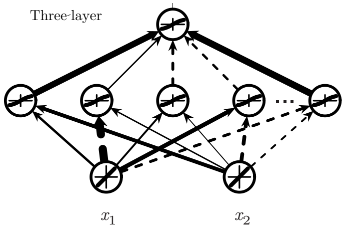
Linear and nonlinear networks

- If a network has no hidden units, then
$$f_i(x) = \phi_i(\langle w_i, x \rangle)$$
That means: f is a linear functions, except perhaps for the final application of ϕ.
- For example: In a classification problem, a two layer network can only represent linear decision boundaries.
- Networks with at least one hidden layer can represent nonlinear decision surfaces.

Two vs Three Layers







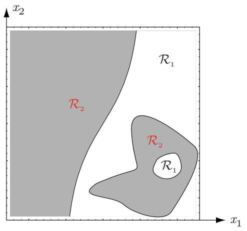
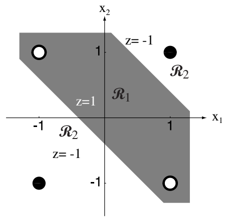
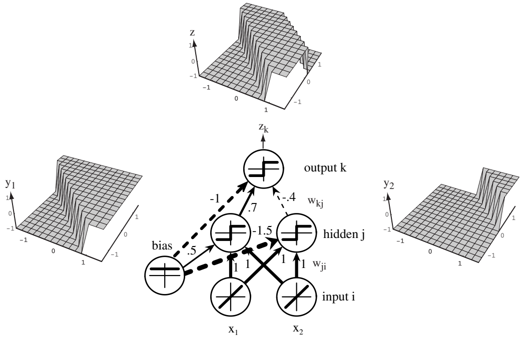


Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

The XOR Problem



Solution regions we would like to represent



Neural network representation

- Two ridges at different locations are substracted from each other.
- That generates a region bounded on both sides.
- A linear classifier cannot represent this decision region.
- Note this requires at least one hidden layer.

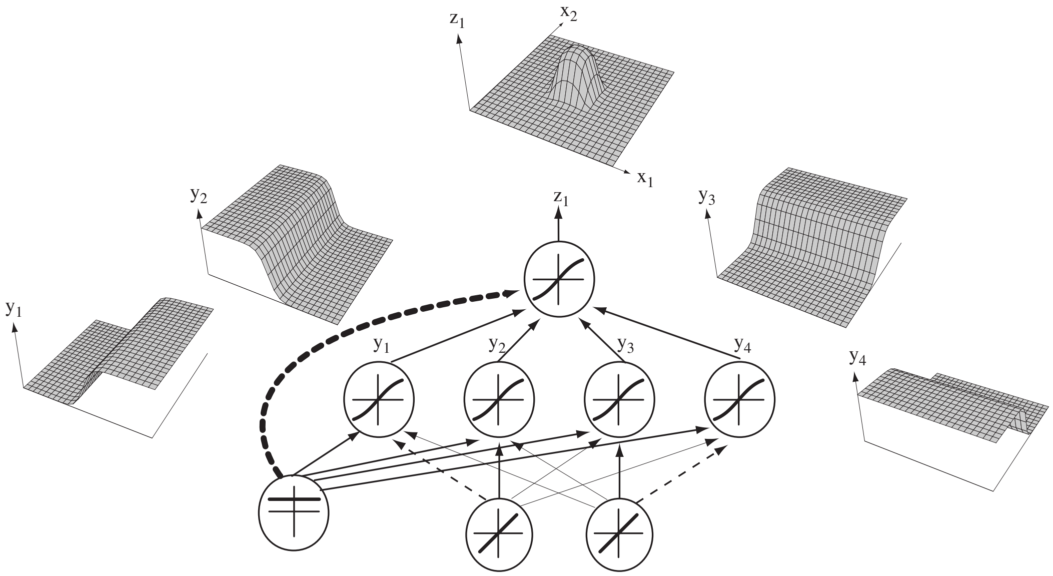


Illustration: R.O. Duda, P.E. Hart, D.G. Stork, *Pattern Classification*, Wiley 2001

Number of Layers

We have observed

- We have seen that two-layer classification networks always represent linear class boundaries.
- With three layers, the boundaries can be non-linear.

Obvious question

- What happens if we use more than three layers? Do four layers again increase expressive power?

Width vs Depth

A neural network represents a (typically) complicated function f by simple functions $\phi_i^{(k)}$.

What functions can be represented?

A well-known result in approximation theory says: Every continuous function $f : [0,1]^d \to \mathbb{R}$ can be represented in the form

$$f(x) = \sum_{j=1}^{2d+1} \xi_j \Bigl(\sum_{i=1}^{d} \tau_{ij}(x_i) \Bigr)$$

where ξ_i and τ_{ij} are functions $\mathbb{R} \to \mathbb{R}$. A similar result shows one can approximate f to arbitrary precision using specifically sigmoids, as

$$f(x) \approx \sum_{j=1}^{M} w_j^{(2)} \sigma \Bigl(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + c_i \Bigr)$$

for some finite M and constants c_i.

Note the representations above can both be written as neural networks with three layers (i.e. with one hidden layer).

Width vs Depth

Depth rather than width

- The representations above can achieve arbitrary precision with a single hidden layer (roughly: a three-layer neural network can represent any continuous function).
- In the first representation, ξ_j and τ_{ij} are "simpler" than f because they map $\mathbb{R} \to \mathbb{R}$.
- In the second representation, the functions are more specific (sigmoids), and we typically need more of them (M is large).
- That means: The price of precision is many hidden units, i.e. the network grows wide.
- The last years have shown: We can obtain very good results by limiting layer width, and instead increasing depth (= number of layers).
- Theory is starting to emerge to properly explain this behavior.

(see e.g. Pleiss and Cunningham 2021 NeurIPS)

Limiting width

- Limiting layer width means we limit the degrees of freedom of each function $f^{(k)}$.
- That is a notion of parsimony.

...hence "Deep Learning"

TRAINING NEURAL NETWORKS

Task

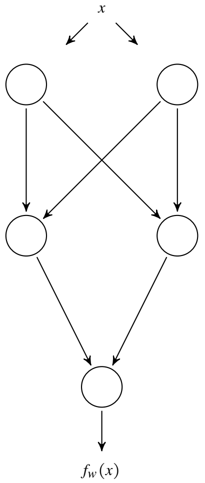
- We decide on a neural network "architecture": We fix the network diagram, including all functions ϕ at the units. Only the weights w on the edges can be changed during by training algorithm. Suppose the architecture we choose has d_1 input units and d_2 output units.
- We collect all weights and biases into a vector θ. The entire network then represents a function $f_\theta(x)$ that maps $\mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$.
- To "train" the network now means that, given training data, we have to determine a suitable parameter vector w, i.e. we fit the network to data by fitting the weights.

More specifically: Classification

Suppose the network is meant to represent a two-class classifier.

- That means the output dimension is $d_2 = 1$, so f_w is a function $\mathbb{R}^{d_1} \to \mathbb{R}$.
- We are given data $x_1, x_2, \ldots$ with labels $y_1, y_2, \ldots$.
- We split this data into training, validation and test data, according to the requirements of the problem we are trying to solve.
- We then fit the network to the training data.

Training Neural Networks



- We run each training data point x_i through the network f_θ and compare $f_\theta(x_i)$ to y_i to measure the error.
- Recall how gradient descent works: We make "small" changes to θ, and choose the one which decreases the error most. That is one step of the gradient scheme.
- For each such changed value θ', we again run each training data point x_i through the network $f_{\theta'}$, and measure the error by comparing $f_{\theta'}(x_i)$ to y_i. This is our loss $L(y_i, x_i)$.

Training Neural Networks

Loss function

- We have to specify how we compare the network's output $f_\theta(x)$ to the correct answer y.
- To do so, we specify a function L with two arguments that serves as an error measure.
- The choice of L depends on the problem.

Typical loss functions

- Classification problem:

$$L(\hat{y}, y) := -\sum_{k=1}^{K} y^k \log \hat{y}^k \qquad \text{(with convention } 0 \log 0 = 0\text{)}$$

- Regression problem:

$$L(\hat{y}, y) := \|y - \hat{y}\|^2$$

Training as an optimization problem

- Given: Training data $(x_1, y_1), \ldots, (x_n, y_n)$ with labels y_i.
- We specify a loss L, and define the total error on the training set – the empirical risk – as

$$\mathcal{R}(\theta) := \sum_{i=1}^{n} L(f_\theta(x_i), y_i)$$

Training problem

In summary, neural network training attempts to solve the optimization problem

$$\theta^* = \arg\min_{\theta} \mathcal{R}(\theta)$$

using gradient descent. For feed-forward networks, the gradient descent algorithm takes a specific form that is called *backpropagation*.

Backpropagation is gradient descent applied to $\mathcal{R}(\theta)$ in a feed-forward network.

In practice (foreshadowing): Stochastic gradient descent

- The vector θ can be very high-dimensional. In high dimensions, computing a gradient is computationally expensive, because we have to make "small changes" to θ in many different directions and compare them to each other.
- Each time the gradient algorithm computes $\mathcal{R}(\theta')$ for a changed value θ', we have to apply the network to every data point, since $\mathcal{R}(\theta') = \sum_{i=1}^{n} L(f_{\theta'}(x_i), y_i)$.
- To save computation, the gradient algorithm typically computes $L(f_{\theta'}(x_i), y_i)$ only for some small subset of a the training data. This subset is called a *mini batch*, and the resulting algorithm is called **stochastic gradient descent**.

BACKPROPAGATION

Neural network training optimization problem

$$\min_\theta \mathcal{R}(\theta)$$

The application of gradient descent to this problem is called *backpropagation*.

Backpropagation is gradient descent applied to $\mathcal{R}(\theta)$ in a feed-forward network.

Deriving backpropagation

- We have to evaluate the derivative $\nabla_\theta \mathcal{R}(\theta)$.
- Since $\mathcal{R}$ is additive over training points, $\mathcal{R}(\theta) = \sum_i L(f_\theta(x_i), y_i)$, it suffices to derive $\nabla_\theta L(f_\theta(x_i), y_i)$.

Chain Rule

Recall from calculus: Chain rule

Consider a composition of functions $f \circ g(x) = f(g(x))$.

$$\frac{d(f \circ g)}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

If the derivatives of f and g are f' and g', that means: $\frac{d(f \circ g)}{dx}(x) = f'(g(x))g'(x)$

Application to feed-forward network

Let $\theta^{(k)}$ denote the weights in layer k. The function represented by the network is

$$f_\theta(x) \;=\; f_\theta^{(K)} \circ \cdots \circ f_\theta^{(1)}(x) \;=\; f_{\theta^{(K)}}^{(K)} \circ \cdots \circ f_{\theta^{(1)}}^{(1)}(x)$$

To solve the optimization problem, we have to compute derivatives of the form

$$\frac{d}{d\theta}L(f_\theta(x_i), y_i) = \frac{dL(\,\bullet\,, y_i)}{df_\theta}\frac{df_\theta}{d\theta}$$

Decomposing the Derivatives

- The chain rule means we compute the derivatives layer by layer.
- Suppose we are only interested in the weights of layer k, and keep all other weights fixed. The function f represented by the network is then

$$f_{\theta^{(k)}}(x) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\theta^{(k)}} \circ f^{(k-1)} \circ \cdots \circ f^{(1)}(x)$$

- The first $k-1$ layers enter only as the function value of x, so we define

$$z^{(k)} := f^{(k-1)} \circ \cdots \circ f^{(1)}(x)$$

  and get

$$f_{\theta^{(k)}}(x) \;=\; f^{(K)} \circ \cdots \circ f^{(k+1)} \circ f^{(k)}_{\theta^{(k)}}(z^{(k)})$$

- If we differentiate with respect to $\theta^{(k)}$, the chain rule gives

$$\frac{d}{d\theta^{(k)}} f_{\theta^{(k)}}(x) \;=\; \frac{df^{(K)}}{df^{(K-1)}} \cdots \frac{df^{(k+1)}}{df^{(k)}} \cdot \frac{df^{(k)}_{\theta^{(k)}}}{d\theta^{(k)}}$$

Within a Single Layer

- Each $f^{(k)}$ is a vector-valued function $f^{(k)} : \mathbb{R}^{d_k} \to \mathbb{R}^{d_{k+1}}$.
- It is parametrized by the weights $\theta^{(k)}$ of the kth layer and takes an input vector $z \in \mathbb{R}^{d_k}$.
- We write $f^{(k)}(z, \theta^{(k)})$.

Layer-wise derivative

Since $f^{(k)}$ and $f^{(k-1)}$ are vector-valued, we get a Jacobian matrix

$$\frac{df^{(k+1)}}{df^{(k)}} = \begin{pmatrix} \frac{\partial f_1^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_1^{(k+1)}}{\partial f_{d_k}^{(k)}} \\ \vdots & & \vdots \\ \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_1^{(k)}} & \cdots & \frac{\partial f_{d_{k+1}}^{(k+1)}}{\partial f_{d_k}^{(k)}} \end{pmatrix} =: \Delta^{(k)}(z, \theta^{(k+1)})$$

- $\Delta^{(k)}$ is a matrix of size $d_{k+1} \times d_k$.
- The derivatives in the matrix quantify how $f^{(k+1)}$ reacts to changes in the argument of $f^{(k)}$ if the weights $\theta^{(k+1)}$ and $\theta^{(k)}$ of both functions are fixed.

Backpropagation algorithm

Let $\theta^{(1)}, \dots, \theta^{(K)}$ be the current settings of the layer weights. These have either been computed in the previous iteration, or (in the first iteration) are initialized at random.

Step 1: Forward pass

We start with an input vector x and compute

$$z^{(k)} := f^{(k)} \circ \cdots \circ f^{(1)}(x)$$

for all layers k.

Step 2: Backward pass

- Start with the last layer. Update the weights $\theta^{(K)}$ by performing a gradient step on

  $$L\big(f^{(K)}(z^{(K)}, \theta^{(K)}), y\big)$$

  regarded as a function of $\theta^{(K)}$ (so $z^{(K)}$ and y are fixed). Denote the updated weights $\tilde{\theta}^{(K)}$.
- Move backwards one layer at a time. At layer k, we have already computed updates $\tilde{\theta}^{(K)}, \dots, \tilde{\theta}^{(k+1)}$. Update $\theta^{(k)}$ by a gradient step, where the derivative is computed as

  $$\Delta^{(K-1)}(z^{(K-1)}, \tilde{\theta}^{(K)}) \cdot \ldots \cdot \Delta^{(k)}(z^{(k)}, \tilde{\theta}^{(k+1)}) \frac{df^{(k)}}{d\theta^{(k)}}(z, \theta^{(k)})$$

On reaching level 1, go back to step 1 and recompute the $z^{(k)}$ using the updated weights.

SUMMARY: BACKPROPAGATION

- Backpropagation is a gradient descent method for the optimization problem

$$\min_{\theta} \mathcal{R}(\theta) = \sum_{i=1}^{N} L(f_\theta(x_i), y_i)$$

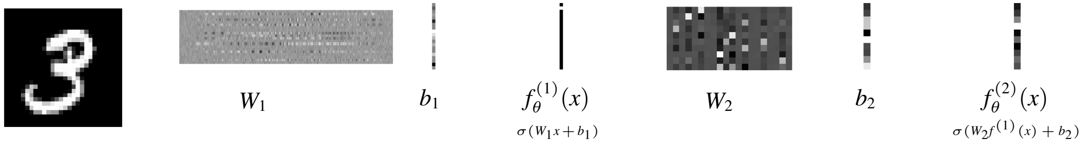
  L must be chosen such that it is additive over data points.
- It alternates between forward passes that update the layer-wise function values $z^{(k)}$ given the current weights, and backward passes that update the weights using the current $z^{(k)}$.
- The layered architecture means we can (1) compute each $z^{(k)}$ from $z^{(k-1)}$ and (2) we can use the weight updates computed in layers $K, \dots, k+1$ to update weights in layer k.

So that's great, but implementing these steps seems hard and tedious...

Convolutional Neural Networks

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

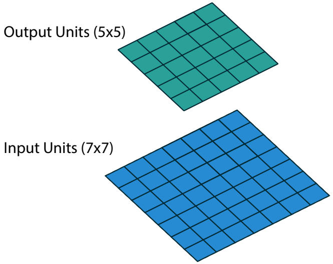
Opportunity:

- What dependency does x_1 have on x_{784}? x_2? x_{29}?
- 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): $\mathbb{R}^{49 \times 25} \to |\theta| = 1225$



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

- New weight matrix: $\mathbb{R}^{3 \times 3} \to |\theta| = 9$
- $> 100\times$ savings in parameters!
- But we have lost expressivity...

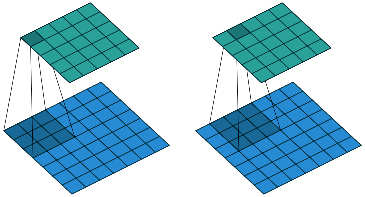
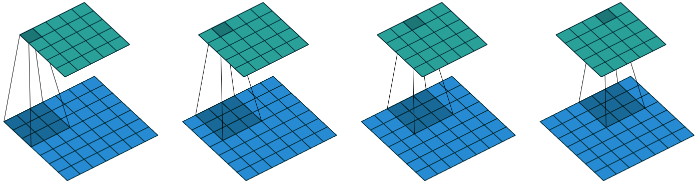


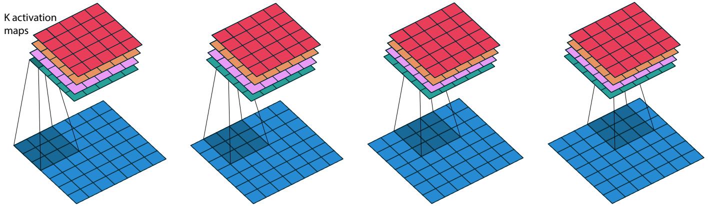
Image credit for all of these and the following: `https://github.com/vdumoulin/conv_arithmetic`

CONVOLUTIONAL LAYER

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



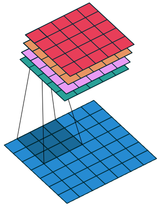
Now use multiple filters (below $K = 4$), producing multiple *activation maps* (each 5×5)



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

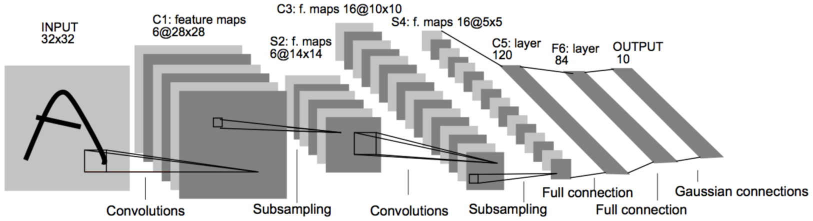
Our/tf convention for 2D convolution: filter width $\times$ filter height $\times$ input depth $\times$ output depth.

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:



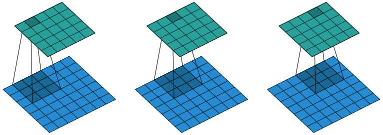
[LeCun et al 1998]

- 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:

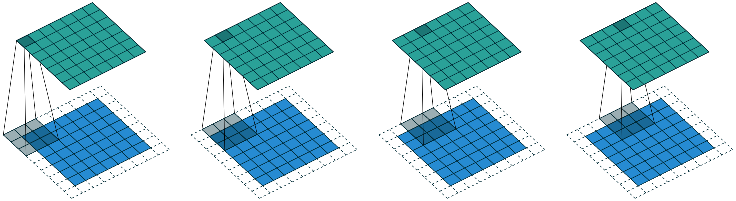
- Filtering reduces spatial extent of activation map
- Edge pixels/activations are less frequently seen
- (Note these can also be benefits)



Zero Padding:

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

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

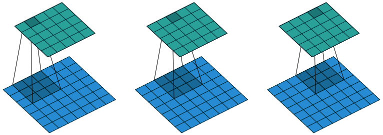


Note: one can zero-pad more/less/asymetrically/otherwise, with varied problem-specific effects

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

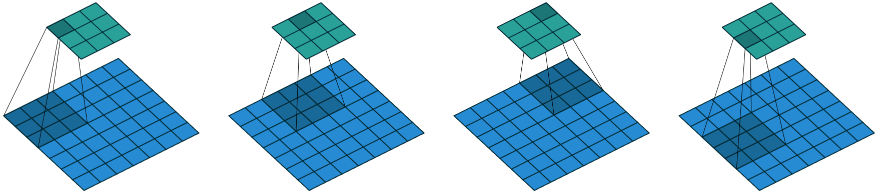


Stride:

- 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.

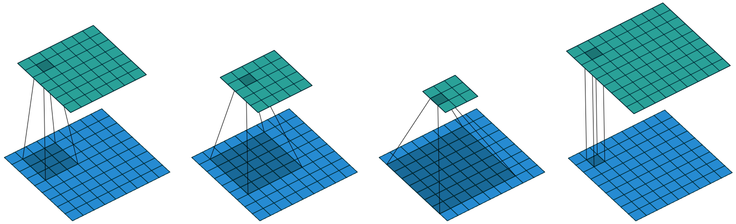


Note: striding and zero-padding give design flexibility and balance each other

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?

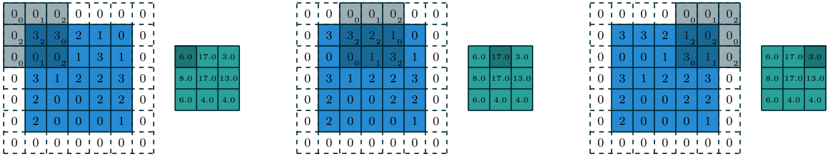
- No! Remember, the conv layer is filter width $\times$ filter height $\times$ input depth $\times$ output depth
- Critical: **filters always operate on the whole depth of the input activation stack**
- 1×1 conv layers $\rightarrow$ dimension reduction: preserve map size, reduce output dimension K

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 ) + b )`
- Compare to `tf.nn.relu( tf.matmul( x , W ) + b)`

Specific example

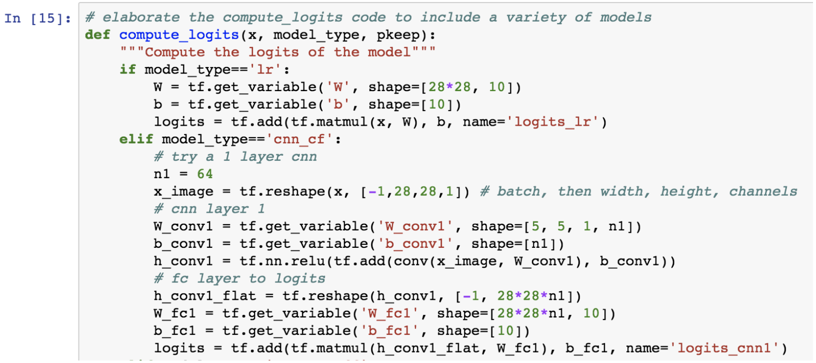


Questions

- 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



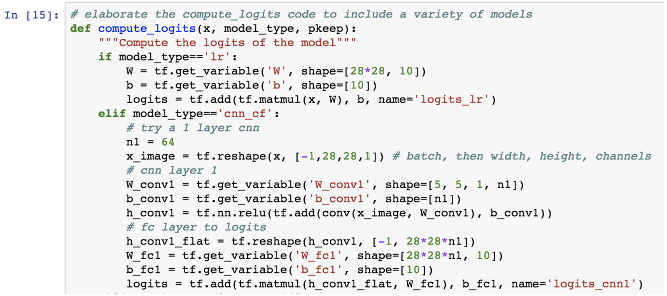
```
In [15]: # elaborate the compute_logits code to include a variety of models
         def compute_logits(x, model_type, pkeep):
             """Compute the logits of the model"""
             if model_type=='lr':
                 W = tf.get_variable('W', shape=[28*28, 10])
                 b = tf.get_variable('b', 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:

- *logits* are the real-valued inputs to the final nonlinear (softmax) transformation.
- This network should be more expressive than logistic regression
- Compare $|\theta|$ with logistic regression
- (this code is lower-level than we will need...)

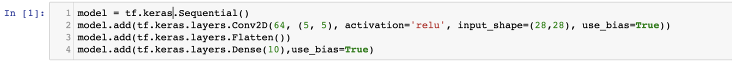
In Practice: from `tf` to `keras`

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')
```

Compare to:



```
In [1]: model = tf.keras.Sequential()
model.add(tf.keras.layers.Conv2D(64, (5, 5), activation='relu', input_shape=(28,28), use_bias=True))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(10),use_bias=True)
```

Keras:

- ...is a high-level API that is now (almost) fully integrated into tensorflow.
- ...is what many of you will use in your projects.
- ...is quite a bit easier than direct tensorflow
- ...obscures some key didactic details, so we will go back and forth in presentation

GENERALIZING THE LOGISTIC MAP

We need to map continuous outputs to a set of K probabilities (in fact, the K-simplex):

$$\mathrm{softmax}(x)^j = \frac{e^{x^j}}{\sum_{k=1}^{K} e^{x^k}}$$

Cross-entropy loss, with a one-hot encoded label y_i:

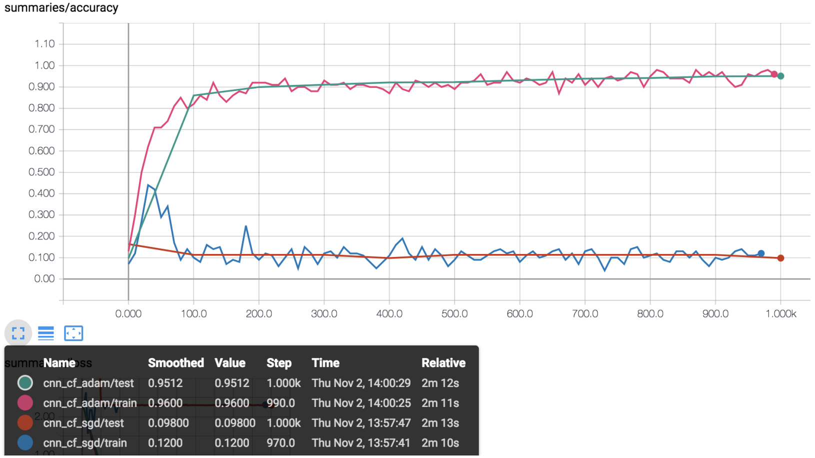
$$L(y_i, f_\theta(x_i)) = -\sum_{k=1}^{K} y_i^k \log f_\theta(x_i)^k$$

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
- Conveniently, `tf` and `keras` obscure most of these details from you
- ...but you will still run into issues at some point

CAUTION: CHOICE OF OPTIMIZER

Consider different SGD variants (much more on SGD in subsequent lectures)

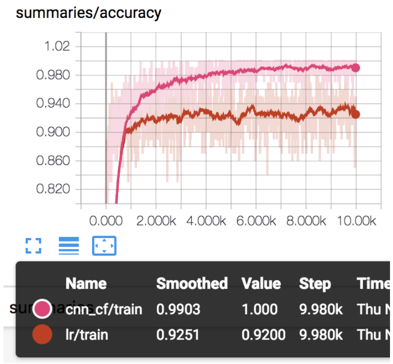


	Name	Smoothed	Value	Step	Time	Relative
●	cnn_cf_adam/test	0.9512	0.9512	1.000k	Thu Nov 2, 14:00:29	2m 12s
●	cnn_cf_adam/train	0.9600	0.9600	990.0	Thu Nov 2, 14:00:25	2m 11s
●	cnn_cf_sgd/test	0.09800	0.09800	1.000k	Thu Nov 2, 13:57:47	2m 13s
●	cnn_cf_sgd/train	0.1200	0.1200	970.0	Thu Nov 2, 13:57:41	2m 10s

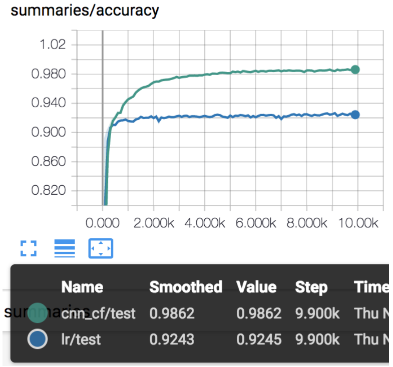
We will stick mostly with Adam for remainder, but again, empiricism...

Progress with cnn_cf

Training and Test



	Name	Smoothed	Value	Step	Time
●	cnn_cf/train	0.9903	1.000	9.980k	Thu N
●	lr/train	0.9251	0.9200	9.980k	Thu N



	Name	Smoothed	Value	Step	Time
●	cnn_cf/test	0.9862	0.9862	9.900k	Thu N
●	lr/test	0.9243	0.9245	9.900k	Thu N

Questions

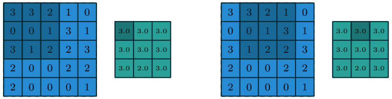
- Why is test/train nonsmooth/smooth?
- How do I set up tensorboard summaries for train and test?
- Will we do better if we make this network more complicated/deeper?
- Am I concerned by a $\approx 0.4\%$ difference between train and test?

TRICKS OF THE TRADE: POOLING

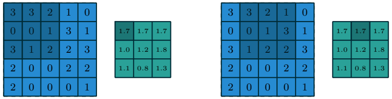
Idea

- 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

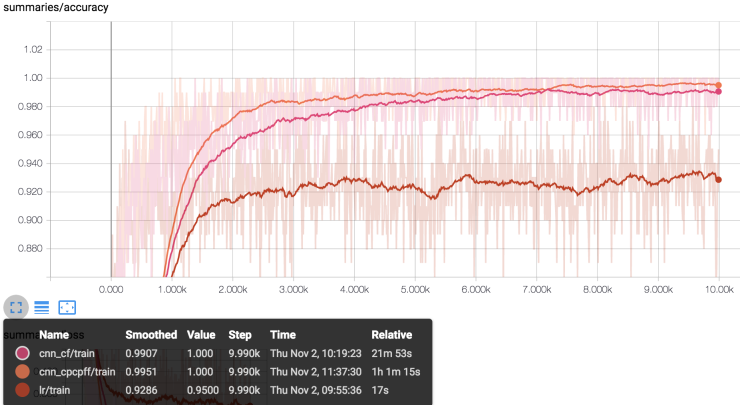


Now

- 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`



	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

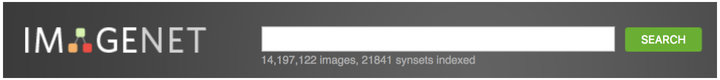
Worth it?

- Better, but not much better.
- More costly

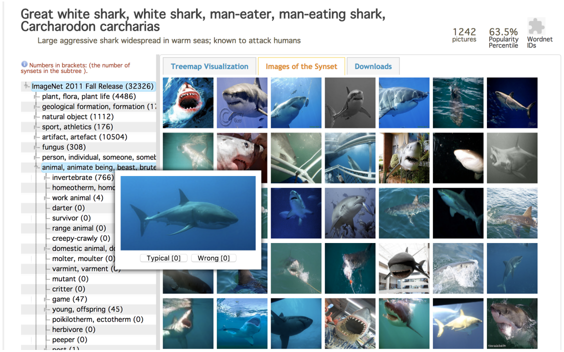
This story will change with more complex datasets...

IMAGENET

The textbook large-scale vision dataset



14,197,122 images, 21841 synsets indexed



Great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias

Large aggressive shark widespread in warm seas; known to attack humans

1242 pictures | 63.5% Popularity Percentile | Wordnet IDs

Numbers in brackets: (the number of synsets in the subtree).

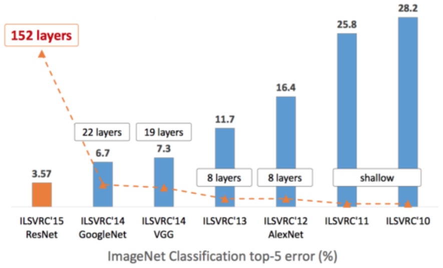
- ImageNet 2011 Fall Release (32326)
  - plant, flora, plant life (4486)
  - geological formation, formation (17
  - natural object (1112)
  - sport, athletics (176)
  - artifact, artefact (10504)
  - fungus (308)
  - person, individual, someone, someb
  - animal, animate being, beast, brute
    - invertebrate (766)
    - homeotherm, homo
    - work animal (4)
    - darter (0)
    - survivor (0)
    - range animal (0)
    - creepy-crawly (0)
    - domestic animal, d
    - molter, moulter (0)
    - varmint, varment (0)
    - mutant (0)
    - critter (0)
    - game (47)
    - young, offspring (45)
    - poikilotherm, ectotherm (0)
    - herbivore (0)
    - peeper (0)
    - pest (1)

ImageNet Challenge

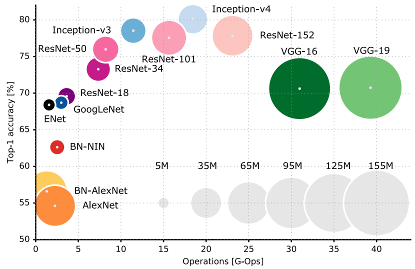
ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

- 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...

Beyer et al (2020) "Are we done with ImageNet?"



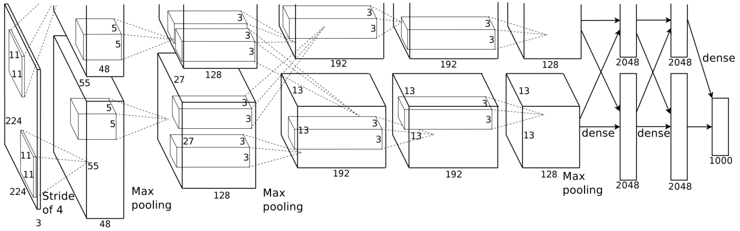
[Kaiming He]



[Canziani et al 2017]

AlexNet

The first ILSVRC winner with deep learning

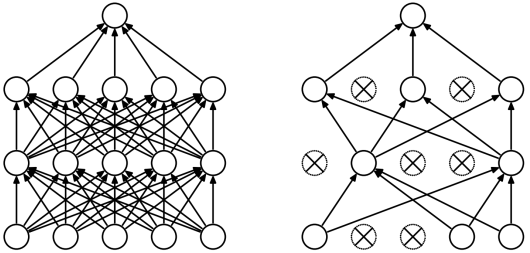


[Krizhevsky et al 2012]

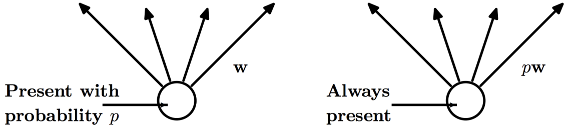
We can understand the entirety of this network

TRICKS OF THE TRADE: DROPOUT

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



(a) Standard Neural Net (b) After applying dropout.



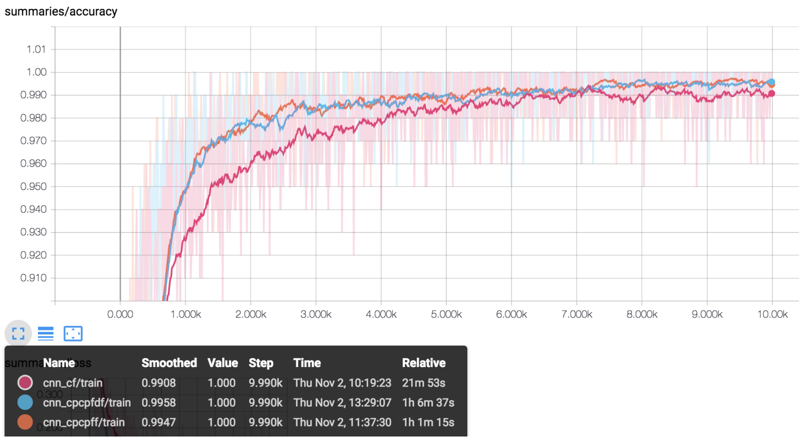
(a) At training time (b) At test time

[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`

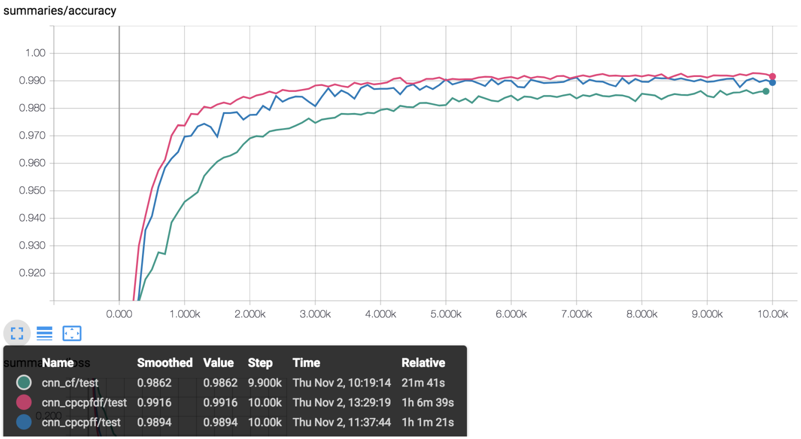


Name	Smoothed	Value	Step	Time	Relative
● cnn_cf/train	0.9908	1.000	9.990k	Thu Nov 2, 10:19:23	21m 53s
● cnn_cpcpfdf/train	0.9958	1.000	9.990k	Thu Nov 2, 13:29:07	1h 6m 37s
● cnn_cpcpff/train	0.9947	1.000	9.990k	Thu Nov 2, 11:37:30	1h 1m 15s

Does not seem to affect training much...

Tricks of the trade: dropout

But hopefully it mitigates overfitting

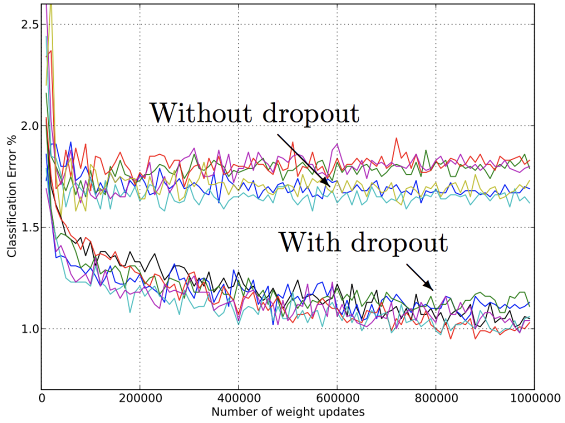


	Name	Smoothed	Value	Step	Time	Relative
●	cnn_cf/test	0.9862	0.9862	9.900k	Thu Nov 2, 10:19:14	21m 41s
●	cnn_cpcpfdf/test	0.9916	0.9916	10.00k	Thu Nov 2, 13:29:19	1h 6m 39s
●	cnn_cpcpff/test	0.9894	0.9894	10.00k	Thu Nov 2, 11:37:44	1h 1m 21s

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:

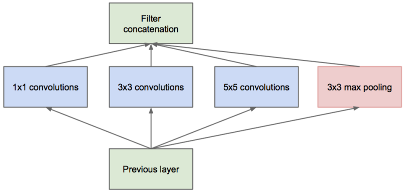
- Change the filters: sizes, striding, padding
- Change the pooling: average/max, different sizes, different positions
- Change the architecture
- Change the optimization method
- Change the batch size
- Change the summary/tensorboard content
- ...

INCEPTION MODULES

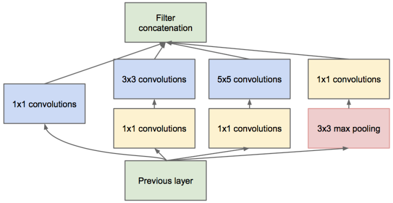
2014 ILSVRC winner added yet more complexity... Idea (for conceptual purposes; don't worry the details):

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

Inception module



(a) Inception module, naïve version

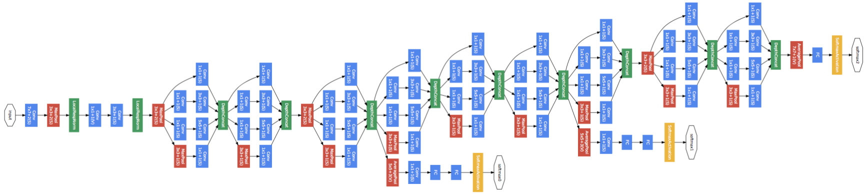


(b) Inception module with dimension reductions

[Szegedy et al 2014]

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

INCEPTION



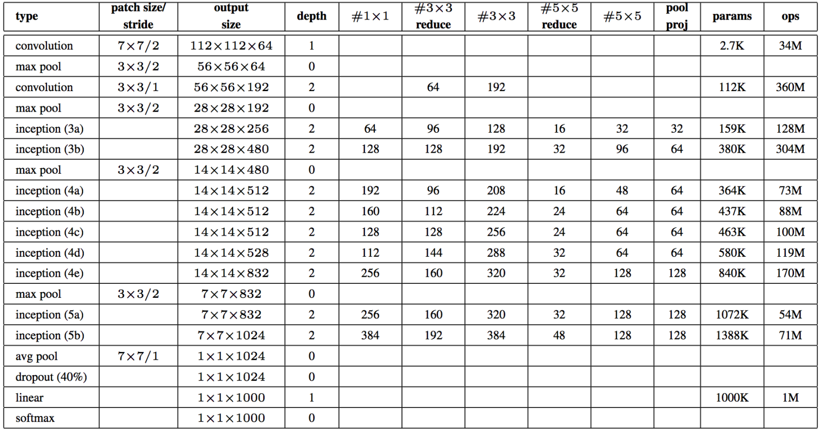
Full network

[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



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								

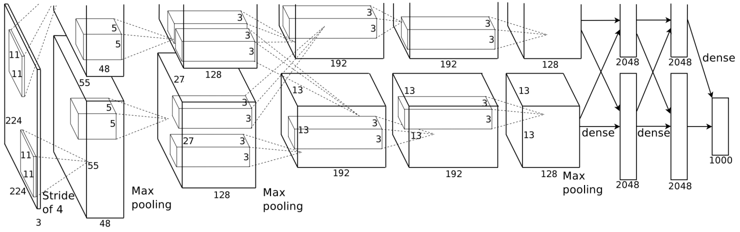
Another view

[Szegedy et al 2014]

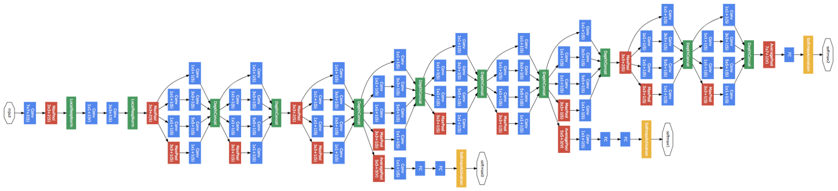
More complex, but still components we understand.

QUICK ASIDE: 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]]

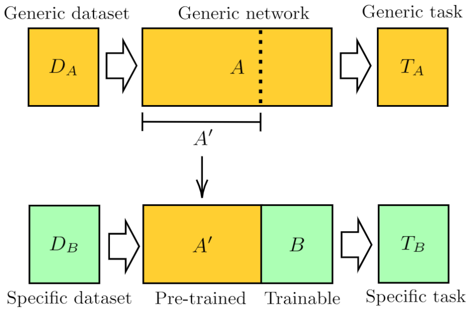
Idea: exploit a large pre-trained network to solve your problem...

QUICK ASIDE: TRANSFER LEARNING

Consider a network as having two stages:

- a feature extractor: many layers that extract a useful representation
- a classifier: a logistic regression to the output of interest
- (note the above is very hand wavy, but is useful intuition and fundamental to much of deep learning thought)

Transfer learning: borrow the first stage (ex: *A* – ImageNET; *B* – your own small image dataset)



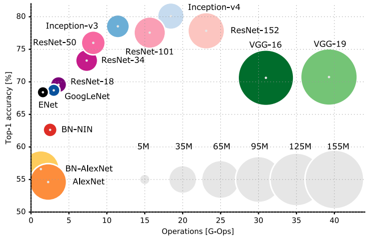
Conceptually transfer learning is easy; the challenge is the code... (see HW02!)

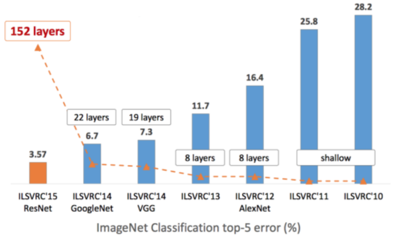
For more, see foundation models like CLIP: [Radford et al 2021]

Returning to performance: 2015 was a big year

Two big (simple) ideas brought the next level of performance:

1. Batch Normalization
2. Residual connections (the 2015 ILSVRC winner)
    - added (vastly) more depth to the network
    - surpassed human level performance
    - did so with reasonably fewer parameters





[Kaiming He], [Canziani et al 2017]

Both of these ideas have become fairly standard practice.

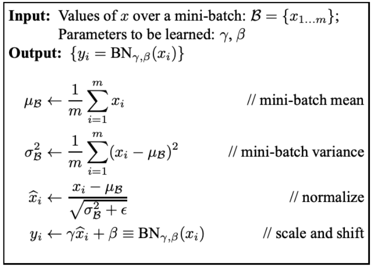
Tricks of the trade: batch norm

Parameter initialization (and learning in general) is made complicated by nonlinearities

- What happens if all inputs are saturated (in say a relu or sigmoid)?
- Distribution of inputs matters (think gradients)!
- *Normalization* layers have been widely used to mitigate.
  - Local response norm.: divide unit activation by sum of squares of local neighbors

[Krizhevsky et al 2012]

  - **Batch normalization**:
    - standardize all units (individually, for compute considerations) across the minibatch to a learned mean and var.
    - γ, β are learned parameters
    - Test time: often an exponentially weighted average of mini-batch batch params



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1 \ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

[Ioffe and Szegedy 2015]

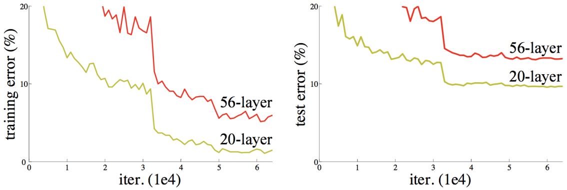
- Batch norm is an important trick of the trade (somewhat replacing dropout and pooling...)

Problems with Depth

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

- Chain rule has multiplicative terms, nonlinearities can saturate, etc.

Degradation has been another key roadblock to increasing depth



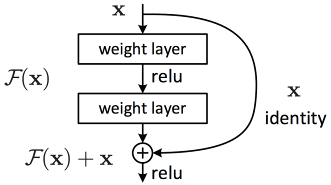
Notice:

- 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...

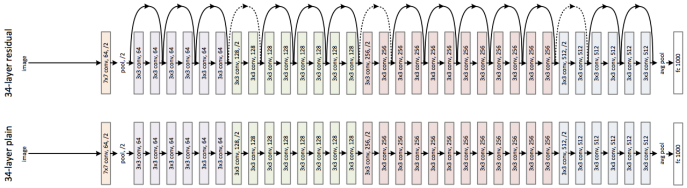
[He et al 2015]

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:



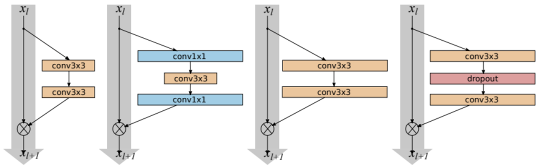
Resulting world leading performance, with many follow-on variations (layer dropout, e.g.)

[He et al 2015]

ResNet

ResNets are still the dominant off-the-shelf architecture choice for computer vision (in 2022)

- Width helps: performance grows slowly in depth (loosely: the skip connections mean that blocks aren't forced to learn anything)
- For CIFAR 10/100: use a WideResNet 28-10 (28 blocks, 10$\times$ as wide)
- For ImageNet (or similar scale): use a WideResNet 50-3
- (also empiricism: conv-BN-relu $\to$ BN-relu-conv)



(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

Figure 1: Various residual blocks used in the paper. Batch normalization and ReLU precede each convolution (omitted for clarity)

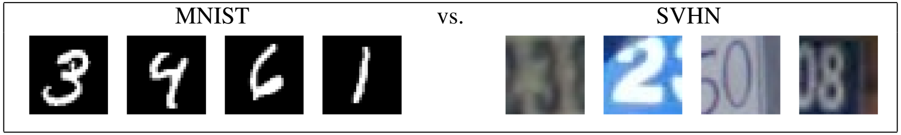
group name	output size	block type = $B(3,3)$
conv1	32×32	[3×3, 16]
conv2	32×32	$\begin{bmatrix} 3\times3, 16\times\text{k} \\ 3\times3, 16\times\text{k} \end{bmatrix} \times\text{N}$
conv3	16×16	$\begin{bmatrix} 3\times3, 32\times\text{k} \\ 3\times3, 32\times\text{k} \end{bmatrix} \times\text{N}$
conv4	8×8	$\begin{bmatrix} 3\times3, 64\times\text{k} \\ 3\times3, 64\times\text{k} \end{bmatrix} \times\text{N}$
avg-pool	1×1	$[8 \times 8]$

[Zagoruyko and Komodakis 2016]

Bleeding edge performance is always changing, but the basic ideas and architectures appear to be in a local optimum.

Back to practicalities: MNIST $\rightarrow$ SVHN

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



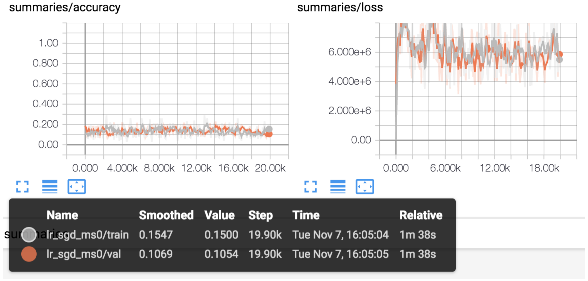
Questions:

- 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. LOGISTIC REGRESSION AND BASIC DEBUGGING

Start with logistic regression and SGD



	Name	Smoothed	Value	Step	Time	Relative
●	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

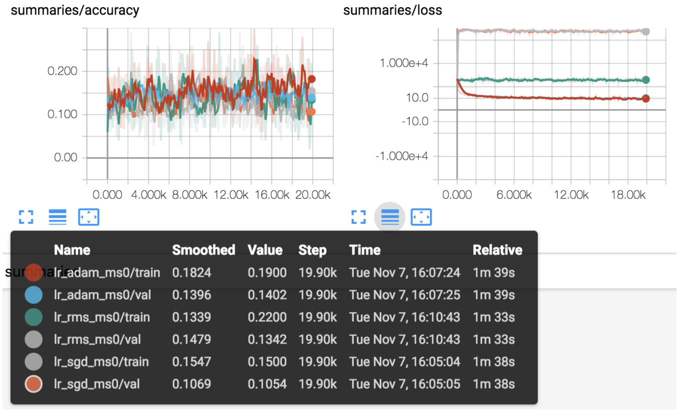
`tb` helps, but basic debugging is still useful

```
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...

2. CHOOSING AN OPTIMIZER

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

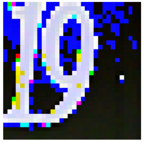


	Name	Smoothed	Value	Step	Time	Relative
●	lr_adam_ms0/train	0.1824	0.1900	19.90k	Tue Nov 7, 16:07:24	1m 39s
●	lr_adam_ms0/val	0.1396	0.1402	19.90k	Tue Nov 7, 16:07:25	1m 39s
●	lr_rms_ms0/train	0.1339	0.2200	19.90k	Tue Nov 7, 16:10:43	1m 33s
●	lr_rms_ms0/val	0.1479	0.1342	19.90k	Tue Nov 7, 16:10:43	1m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

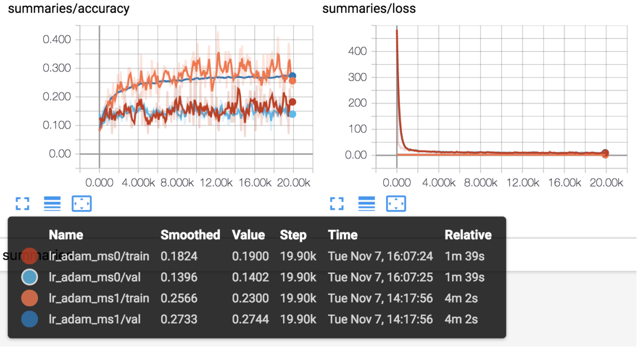
Performance is still terrible, but at least the loss function is not pathological. Progress...

3. MEAN SUBTRACTION

Observation



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

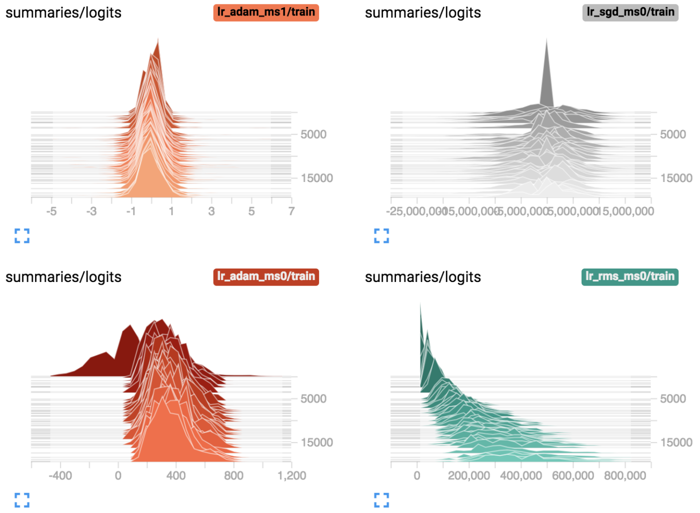


	Name	Smoothed	Value	Step	Time	Relative
●	lr_adam_ms0/train	0.1824	0.1900	19.90k	Tue Nov 7, 16:07:24	1m 39s
●	lr_adam_ms0/val	0.1396	0.1402	19.90k	Tue Nov 7, 16:07:25	1m 39s
●	lr_adam_ms1/train	0.2566	0.2300	19.90k	Tue Nov 7, 14:17:56	4m 2s
●	lr_adam_ms1/val	0.2733	0.2744	19.90k	Tue Nov 7, 14:17:56	4m 2s

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

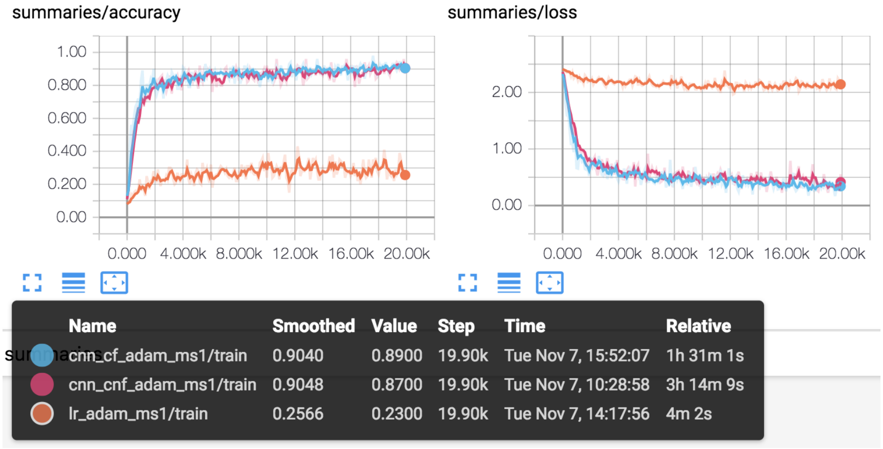
4. TENSORBOARD FOR EMPIRICISM

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



5. ADDING COMPLEXITY

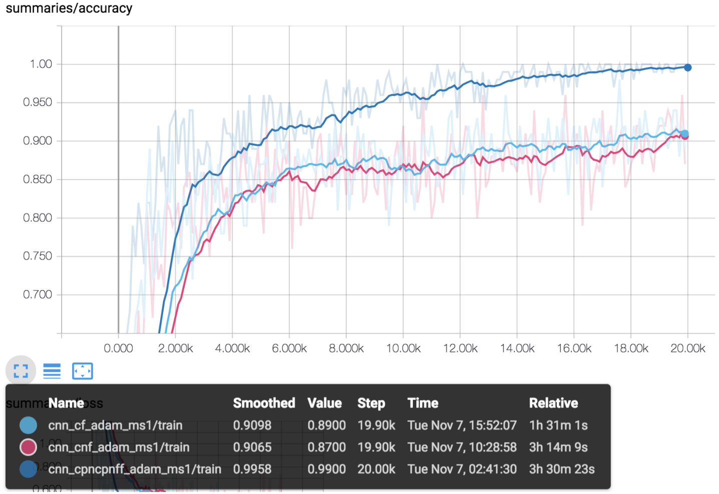
Add `cnn_cf`: `conv` $\rightarrow$ `fc` and `cnn_cnf`: `conv` $\rightarrow$ `norm` $\rightarrow$ `fc`



	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

6. 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

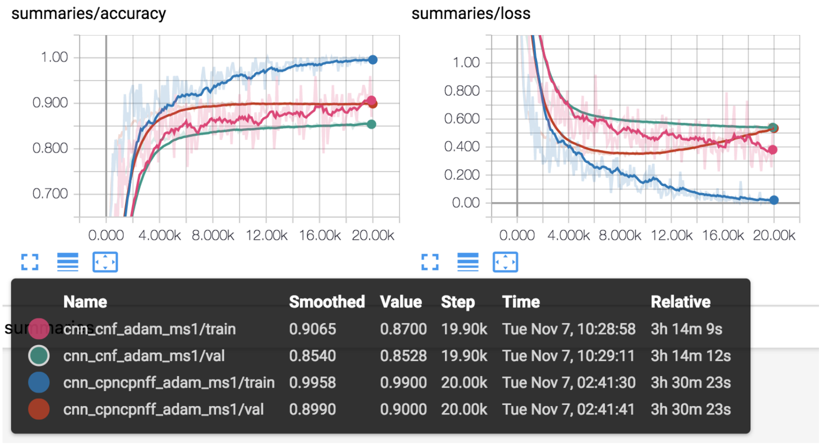
Training performance is very high. Overfitting?

7. VALIDATION DATA

A separate validation set:

- helps monitor training
- avoids data snooping (overfitting to the test set)
- clarifies overfitting (is train/val gap the same as overfitting?)

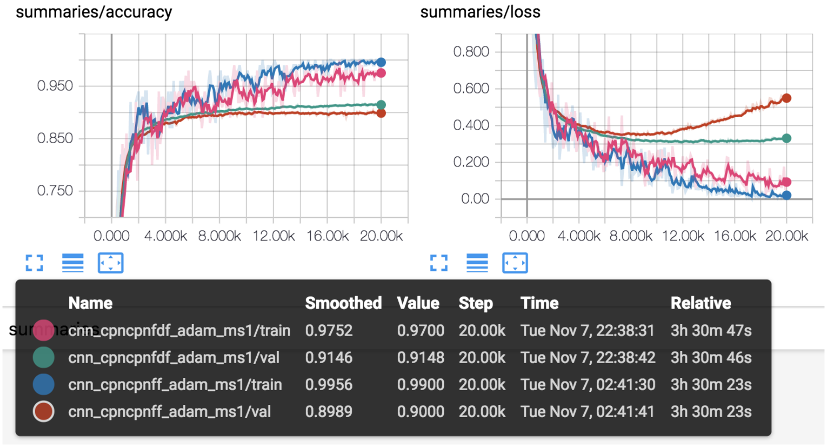
[Bartlett et al (2019) *Benign overfitting in linear regression*]



	Name	Smoothed	Value	Step	Time	Relative
●	cnn_cnf_adam_ms1/train	0.9065	0.8700	19.90k	Tue Nov 7, 10:28:58	3h 14m 9s
●	cnn_cnf_adam_ms1/val	0.8540	0.8528	19.90k	Tue Nov 7, 10:29:11	3h 14m 12s
●	cnn_cpncpnff_adam_ms1/train	0.9958	0.9900	20.00k	Tue Nov 7, 02:41:30	3h 30m 23s
●	cnn_cpncpnff_adam_ms1/val	0.8990	0.9000	20.00k	Tue Nov 7, 02:41:41	3h 30m 23s

8. 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

9. 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.

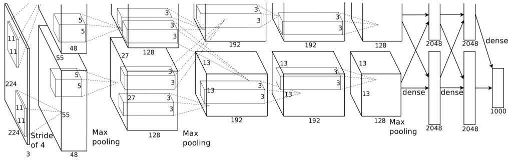
Tools and tricks at your disposal (many more to come):

- 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 one key idea behind 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.?

Next: under the hood of optimization (SGD and autodiff)