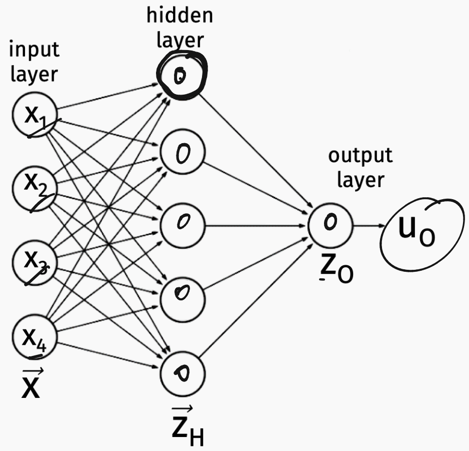
CS-GY 6923: Lecture 11
Backpropagation, Convolutional Neural Networks, Adversarial Examples

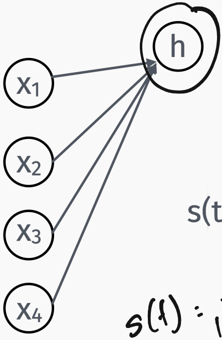
NYU Tandon School of Engineering, Prof. Christopher Musco

Neural networks are a very general family of functions that combine linear "layers" with non-linear activiation functions. Can we used for regression or classification.

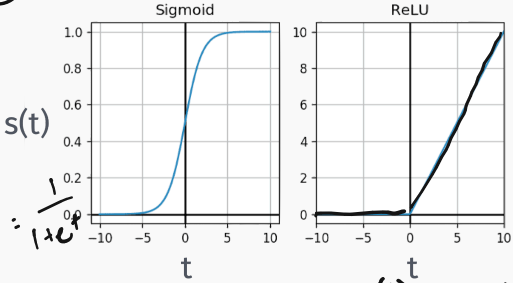


NEURAL NETWORK EQUATIONS

Neural network math:



$$h = \underline{s}(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b)$$



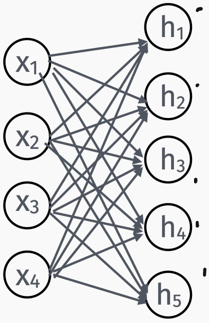
$s(t) = \frac{1}{1+e^{-t}}$

$s(t) = \max(0, t)$

We have one parameter for every edge in the diagram (a weight) and one for every node (a bias).

NEURAL NETWORK EQUATIONS

Usually think about the weights as organized into matrices, one per layer.



$$h_1 = s(w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + w_{1,4}x_4 + b_1)$$

$$h_2 = s(w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 + w_{2,4}x_4 + b_2)$$

$$h_3 = s(w_{3,1}x_1 + w_{3,2}x_2 + w_{3,3}x_3 + w_{3,4}x_4 + b_3)$$

$$h_4 = s(w_{4,1}x_1 + w_{4,2}x_2 + w_{4,3}x_3 + w_{4,4}x_4 + b_4)$$

$$h_5 = s(w_{5,1}x_1 + w_{5,2}x_2 + w_{5,3}x_3 + w_{5,4}x_4 + b_5)$$

NEURAL NETWORK EQUATIONS

Usually think about the weights as organized into matrices, one per layer.

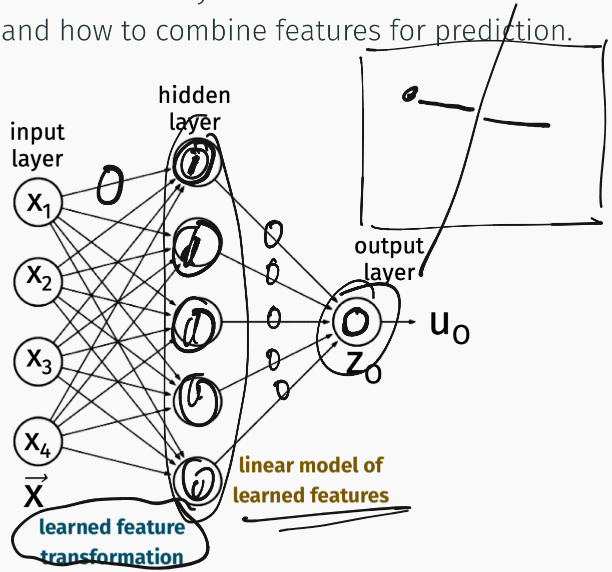
$5 \times 4 \times 4 \times 1 \rightarrow 5 \times 1$

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \end{bmatrix} = \boxed{s}\left[\begin{bmatrix} \underline{W_{1,1}} & \underline{W_{1,2}} & \underline{W_{1,3}} & \underline{W_{1,4}} \\ W_{2,1} & W_{2,2} & W_{2,3} & W_{2,4} \\ W_{3,1} & W_{3,2} & W_{3,3} & W_{3,4} \\ W_{4,1} & W_{4,2} & W_{4,3} & W_{4,4} \\ W_{5,1} & W_{5,2} & W_{5,3} & W_{5,4} \end{bmatrix} \begin{bmatrix} \underline{x_1} \\ \underline{x_2} \\ \underline{x_3} \\ \underline{x_4} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \right]$$

This is also how computations are arranged: very fast to compute matrix-multiplications on GPUs.

MAIN IDEA OF NEURAL NETWORKS

Neural networks simultaneously learn a feature transformation, and how to combine features for prediction.



TRAINING NEURAL NETWORKS

$$L(y_i, f(\theta, x_i)) = (y_i - f(\theta, x_i))^2$$

Let $\underline{f(\boldsymbol{\theta}, \mathrm{x})}$ be our neural network.

Goal: Given training data $(\mathrm{x}_1, y_1), \ldots, (\mathrm{x}_n, y_n)$ minimize the loss

$$\underline{\mathcal{L}(\boldsymbol{\theta})} = \underline{\sum_{i=1}^{n} \underline{L}\left(\underline{y_i}, f\underline{(\boldsymbol{\theta}, \mathrm{x}_i)}\right)},$$

where L is, e.g., binary cross-entropy (logistic) loss for classification, ℓ_2 loss for regression, etc.

GRADIENT OF THE LOSS

Approach: minimize the loss by using stochastic gradient descent.

So we can focus on computing the gradient for a single training example $\underline{(\mathrm{x}, y)}$:

$$\underline{\nabla L\left(y, f(\boldsymbol{\theta}, \mathrm{x})\right).}$$

MULTIVARIABLE CHAIN RULE

Let $y(\underline{x}), z(\underline{x}), w(\underline{x})$ be functions of $\underline{x}$ and let $\underline{f}(\underline{y}, \underline{z}, \underline{w})$ be a function of y, z, w.

$$\left(\frac{df}{dx}\right) = \frac{df}{dy} \cdot \frac{dy}{dx} + \underbrace{\frac{df}{dz} \cdot \frac{dz}{dx}} + \frac{df}{dw} \cdot \frac{dw}{dx}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \cdot \frac{\partial y}{\partial x} + \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x} + \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial x}$$

GRADIENT OF THE LOSS

Applying chain rule each partial derivative of the loss:

$$\nabla L\left(y, f(\boldsymbol{\theta}, \mathrm{x})\right) = \left(\frac{\partial L}{\partial f(\boldsymbol{\theta}, \mathrm{x})}\right) \nabla f(\boldsymbol{\theta}, \mathrm{x})$$

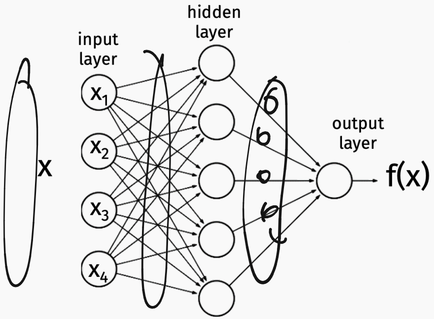
Binary cross-entropy example:

$$L\left(y, f(\boldsymbol{\theta}, \mathrm{x})\right) = -y \log(f(\boldsymbol{\theta}, \mathrm{x})) - (1-y)\log(1 - f(\boldsymbol{\theta}, \mathrm{x}))$$

$$\frac{\partial L}{\partial f(\theta, x)} = -y \frac{1}{f(\theta, x)} - (1-y) \frac{1}{1 - f(\theta, x)} \cdot (-1)$$

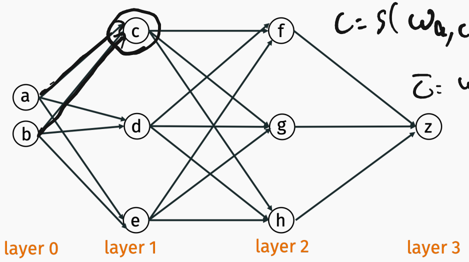
GRADIENT OF THE LOSS

We have reduced our goal to computing $\nabla f(\boldsymbol{\theta}, \mathrm{x})$, where the gradient is with respect to the parameters $\boldsymbol{\theta}$.



Backpropagation: efficient way to compute $\nabla f(\boldsymbol{\theta}, \mathrm{x})$. It derives its name because we compute gradient from back to front: starting with the parameters closest to the output of the neural net.

BACKPROP NOTATION



$c = s(w_{a,c}\, a + w_{b,c}\, b + \beta_c)$

$\bar{c} = w_{a,c}\, a + w_{b,c}\, b \cdot \beta_c$

Notation for few slides:

- $a, b, \ldots, z$ are the node names, and denote values at the nodes after applying non-linearity.
- $\bar{a}, \bar{b}, \bar{c} \ldots, \bar{z}$ denote values before applying non-linearity, but after adding bias.
- $W_{i,j}$ is the weight of edge from node i to node j.
- $s(\cdot) : \mathbb{R} \to \mathbb{R}$ is the non-linear activation function.
- β_j is the bias for node j.

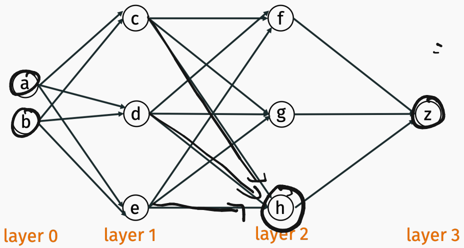
BACKPROP NOTATION

X, y

$x: \{a, b\}$

$$\nabla \sum_{i=1}^{n} L(y_i, f(x_i, \theta))$$

$$= \sum_{i=1}^{n} \nabla L(y_i, f(x_i, \theta))$$



Example: $h = s(c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + \beta_h)$ and

$$\bar{h} = c \cdot W_{c,h} + d \cdot W_{d,h} + e \cdot W_{e,h} + \beta_h.$$

BACKPROP EXAMPLE

Goal: Compute the gradient $\nabla f(\boldsymbol{\theta}, \mathbf{x})$, which contains the partial derivatives with respect to every parameter:

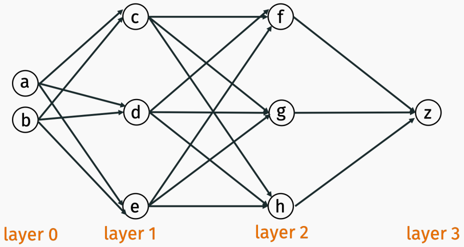
$\frac{\partial z}{\ldots}$

- $\partial z / \partial \beta_z$
- $\partial z / \partial W_{f,z}$, $\partial z / \partial W_{g,z}$, $\partial z / \partial W_{h,z}$
- $\partial z / \partial \beta_f$, $\partial z / \partial \beta_g$, $\partial z / \partial \beta_h$
- $\partial z / \partial W_{c,f}$, $\partial z / \partial W_{c,g}$, $\partial z / \partial W_{c,h}$
- $\partial z / \partial W_{d,f}$, $\partial z / \partial W_{d,g}$, $\partial z / \partial W_{d,h}$
- $\vdots$
- $\partial z / \partial W_{a,c}$, $\partial z / \partial W_{a,d}$, $\partial z / \partial W_{a,e}$

Two steps: (Forward pass) to compute function value.
(Backwards pass) to compute gradients.

BACKPROP EXAMPLE

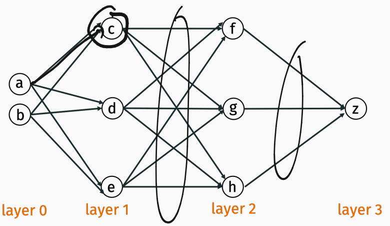
Step 1: Forward pass.



- Using current parameters, compute the output z by moving from left to right.
- Store all intermediate results:

$$\bar{c}, \bar{d}, \bar{e}, c, d, e, \bar{f}, \bar{g}, \bar{h}, f, g, h, \bar{z}, z.$$

BACKPROP EXAMPLE



$\underline{O(P)}$

Step 1: Forward pass.

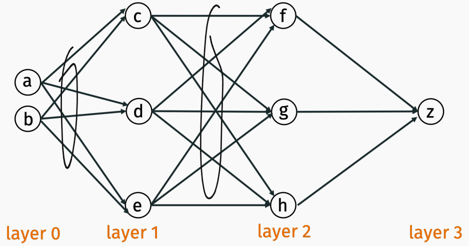
$$\bar{c} = W_{a,c} \cdot a + W_{b,c} \cdot b + \beta_c \qquad c = \underline{s}(\bar{c})$$
$$\bar{d} = W_{a,d} \cdot a + W_{b,d} \cdot b + \beta_d \qquad d = s(\bar{d})$$
$$\bar{e} = W_{a,e} \cdot a + W_{b,e} \cdot b + \beta_e \qquad e = s(\bar{e})$$
$$\bar{f} = W_{c,f} \cdot c + W_{d,f} \cdot d + W_{e,f} \cdot e + \beta_f \qquad f = s(\bar{f})$$
$$\vdots$$
$$\bar{z} = W_{f,z} \cdot f + W_{g,z} \cdot g + W_{h,z} \cdot f + \beta_z \qquad z = s(\bar{z})$$

Question: What is runtime in terms of # of parameters P?

BACKPROP EXAMPLE

$\longleftarrow$

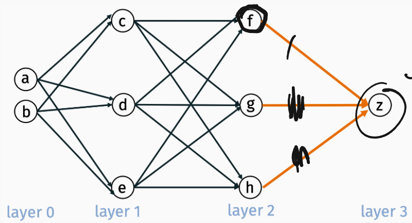
Step 2: Backward pass.



- Using **current parameters** and **computed node values**, compute the partial derivatives of all parameters by moving from right to left.

BACKPROP EXAMPLE

Step 2: Backward pass. Deepest layer.



$z = s(\bar{z})$

$\bar{z} = w_{fz} f + w_{gz} g + w_{hz} h + \beta_z$

$z = s(w_{fz} f + w_{gz} g + w_{hz} h + \beta_z)$

$= s'(w_{fz} f + \dots\dots) \cdot f$

$$\frac{\partial z}{\partial \beta_z} = \frac{\partial \bar{z}}{\partial \beta_z} \cdot \frac{\partial z}{\partial \bar{z}} = \boxed{1 \cdot s'(\bar{z})}$$

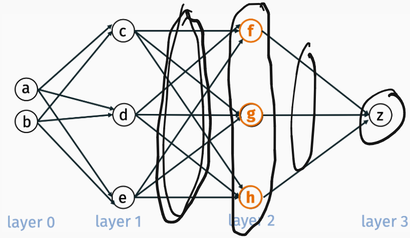
$$\frac{\partial z}{\partial W_{f,z}} = \frac{\partial \bar{z}}{\partial W_{f,z}} \cdot \frac{\partial z}{\partial \bar{z}} = \boxed{f \cdot s'(\bar{z})}$$

$$\frac{\partial z}{\partial W_{g,z}} = \frac{\partial \bar{z}}{\partial W_{g,z}} \cdot \frac{\partial z}{\partial \bar{z}} = g \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial W_{h,z}} = \frac{\partial \bar{z}}{\partial W_{h,z}} \cdot \frac{\partial z}{\partial \bar{z}} = h \cdot s'(\bar{z})$$

BACKPROP EXAMPLE

Step 2: Backward pass.



$$z = s(\bar{z})$$

$$\bar{z} = w_{fz} f + w_{gz} g + w_{hz} h + B_z$$

$$\frac{\partial z}{\partial f} = \frac{\partial \bar{z}}{\partial f} \cdot \frac{\partial z}{\partial \bar{z}} = W_{f,z} \cdot s'(\bar{z})$$

$$\frac{\partial z}{\partial g} = \frac{\partial \bar{z}}{\partial g} \cdot \frac{\partial z}{\partial \bar{z}} = W_{g,z} \cdot s'(\bar{z})$$

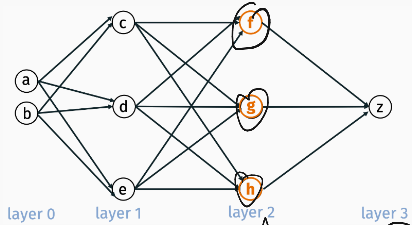
$$\frac{\partial z}{\partial h} = \frac{\partial \bar{z}}{\partial h} \cdot \frac{\partial z}{\partial \bar{z}} = W_{h,z} \cdot s'(\bar{z})$$

Compute partial derivatives with respect to nodes, even though these are not used in the gradient.

BACKPROP EXAMPLE

Step 2: Backward pass.

$f = s(\bar{f})$



$$\frac{\partial z}{\partial \bar{f}} = \boxed{\frac{\partial z}{\partial f}} \cdot \boxed{\frac{\partial f}{\partial \bar{f}}} = \frac{\partial z}{\partial f} \cdot \underline{s'(\bar{f})}$$

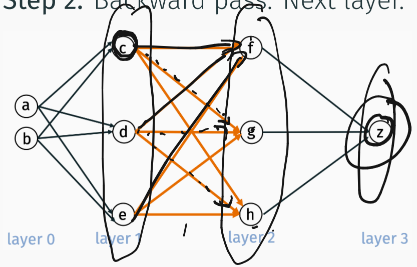
$$\frac{\partial z}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot \frac{\partial g}{\partial \bar{g}} = \frac{\partial z}{\partial g} \cdot s'(\bar{g})$$

$$\frac{\partial z}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot \frac{\partial h}{\partial \bar{h}} = \frac{\partial z}{\partial h} \cdot s'(\bar{h})$$

And for “pre-nonlinearity” nodes.

BACKPROP EXAMPLE

Step 2: Backward pass. Next layer.



$$\bar f = W_{cf}\cdot c + W_{df}\cdot d + W_{ef}\cdot e + B_f$$

$$\frac{dz}{dc} = \frac{\partial z}{\partial f}\cdot\frac{df}{dc}$$

$+ \;. \;. \;.$

$+ \;. \;. \;.$

$$\cancel{\frac{\partial z}{\partial \beta_f} = \frac{\partial z}{\partial \bar f}\cdot\frac{\partial \bar f}{\partial \beta_f}} = \frac{\partial z}{\partial \bar f}\cdot 1$$

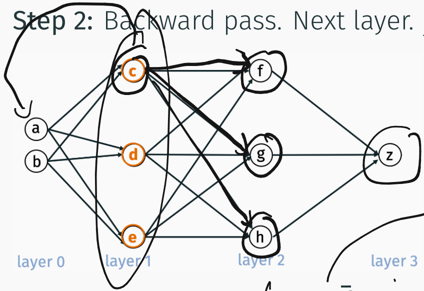
$$\frac{\partial z}{\partial W_{c,f}} = \frac{\partial z}{\underline{\partial \bar f}}\cdot\boxed{\frac{\partial \bar f}{\partial W_{c,f}}} = \frac{\partial z}{\partial \bar f}\cdot \underline{\underline{c}}$$

$$\frac{\partial z}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar f}\cdot\frac{\partial \bar f}{\partial W_{d,f}} = \frac{\partial z}{\partial \bar f}\cdot d$$

$$\frac{\partial z}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar f}\cdot\frac{\partial \bar f}{\partial W_{e,f}} = \frac{\partial z}{\partial \bar f}\cdot e$$

BACKPROP EXAMPLE

Step 2: Backward pass. Next layer. Use multivariate chain rule.



$\bar{f} = w_{cf} \cdot c + \dots$

$\bar{g} = w_{cg} \cdot c + \dots$

$$\frac{\partial z}{\partial c} = \frac{\partial z}{\partial \bar{f}} \cdot \frac{\partial \bar{f}}{\partial c} + \frac{\partial z}{\partial \bar{g}} \cdot \frac{\partial \bar{g}}{\partial c} + \frac{\partial z}{\partial \bar{h}} \cdot \frac{\partial \bar{h}}{\partial c}$$

$$= \frac{\partial z}{\partial \bar{f}} \cdot W_{c,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{c,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{c,h}$$

$$\frac{\partial z}{\partial d} = \frac{\partial z}{\partial \bar{f}} \cdot W_{d,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{d,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{d,h}$$

$$\frac{\partial z}{\partial e} = \frac{\partial z}{\partial \bar{f}} \cdot W_{e,f} + \frac{\partial z}{\partial \bar{g}} \cdot W_{e,g} + \frac{\partial z}{\partial \bar{h}} \cdot W_{e,h}$$

$$\begin{bmatrix} \end{bmatrix} \begin{bmatrix} \partial z/\partial f \\ \vdots \\ \partial z/\partial h \end{bmatrix}$$

BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\underline{\mathbf{v}_i}$ be a vector containing the value of all nodes j in layer i.

$$\underline{\mathbf{v}_3 = \begin{bmatrix} z \end{bmatrix}} \qquad \mathbf{v}_2 = \begin{bmatrix} f \\ g \\ h \end{bmatrix} \qquad \mathbf{v}_1 = \begin{bmatrix} c \\ d \\ e \end{bmatrix}$$

Let $\bar{\mathbf{v}}_i$ be a vector containing $\bar{j}$ for all nodes j in layer i.

$$\underline{\bar{\mathbf{v}}_3} = \begin{bmatrix} \bar{z} \end{bmatrix} \qquad \underline{\bar{\mathbf{v}}_2} = \begin{bmatrix} \bigcirc\bar{f} \\ \bar{g} \\ \bar{h} \end{bmatrix} \qquad \underline{\bar{\mathbf{v}}_1} = \begin{bmatrix} \bar{c} \\ \bar{d} \\ \bar{e} \end{bmatrix}$$

Note: $\mathbf{v}_i = \underline{s}(\bar{\mathbf{v}}_i)$, where s is applied entrywise.

BACKPROP LINEAR ALGEBRA

Linear algebraic view.

Let $\underline{\boldsymbol{\delta}_i}$ be a vector containing $\underline{\partial z/\partial j}$ for all nodes j in layer i.

$$\boldsymbol{\delta}_3 = \begin{bmatrix}1\end{bmatrix} \qquad \boldsymbol{\delta}_2 = \begin{bmatrix}\underline{\partial z/\partial f}\\ \partial z/\partial g\\ \partial z/\partial h\end{bmatrix} \qquad \boldsymbol{\delta}_1 = \begin{bmatrix}\partial z/\partial c\\ \partial z/\partial d\\ \partial z/\partial e\end{bmatrix}$$

Let $\bar{\boldsymbol{\delta}}_i$ be a vector containing $\partial z/\partial \bar{j}$ for all nodes j in layer i.

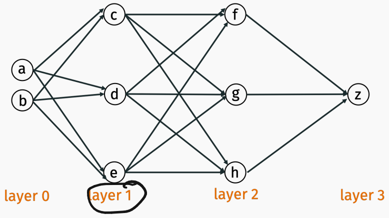
$$\bar{\boldsymbol{\delta}}_3 = \begin{bmatrix}\partial z/\partial \bar{z}\end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_2 = \begin{bmatrix}\boxed{\underline{\partial z/\partial \bar{f}}}\\ \partial z/\partial \bar{g}\\ \partial z/\partial \bar{h}\end{bmatrix} \qquad \bar{\boldsymbol{\delta}}_1 = \begin{bmatrix}\partial z/\partial \bar{c}\\ \partial z/\partial \bar{d}\\ \partial z/\partial \bar{e}\end{bmatrix}$$

Note: $\bar{\boldsymbol{\delta}}_i = s'(\bar{\mathbf{v}}_i) \times \boldsymbol{\delta}_i$ where $\times$ denotes entrywise multiplication.

$$\frac{\partial z}{\partial \bar{f}} = \left(\frac{\partial z}{\partial f}\right)\frac{\partial f}{\partial \bar{f}} \qquad \underline{s'(\bar{f})}$$

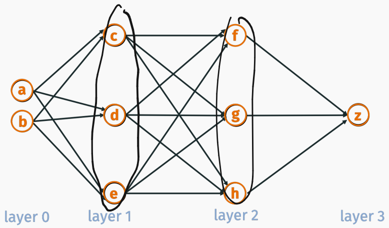
BACKPROP LINEAR ALGEBRA

Let $\mathbf{W}_i$ be a matrix containing all the weights for edges between layer i and layer $i+1$.



$$\mathbf{W}_0 = \begin{bmatrix} W_{a,c} & W_{b,c} \\ W_{a,d} & W_{b,d} \\ W_{a,e} & W_{b,e} \end{bmatrix} \quad \mathbf{W}_1 = \begin{bmatrix} W_{c,f} & W_{d,f} & W_{e,f} \\ W_{c,g} & W_{d,g} & W_{e,g} \\ W_{c,h} & W_{d,h} & W_{e,h} \end{bmatrix} \quad \mathbf{W}_2 = \begin{bmatrix} W_{f,z} & W_{g,z} & W_{h,z} \end{bmatrix}$$

BACKPROP LINEAR ALGEBRA



O(km)
time

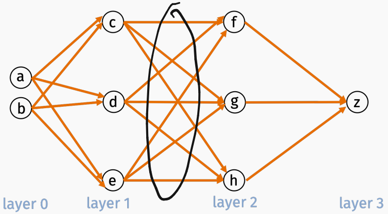
Claim 1: Node derivative computation is matrix multiplication.

$$\underline{\boldsymbol{\delta}_i} = \underline{W_i^T} \underline{\bar{\boldsymbol{\delta}}_{i+1}}$$

What is the computational complexity if $W_i \in \mathbb{R}^{\underline{k} \times \underline{m}}$?

BACKPROP LINEAR ALGEBRA

Let $\boldsymbol{\Delta}_i$ be a matrix contain the derivatives for all weights for edges between layer i and layer $i+1$.

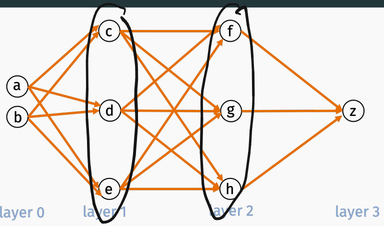


$$\boldsymbol{\Delta}_2 = \begin{bmatrix} \partial z/\partial W_{f,z} & \partial z/\partial W_{g,z} & \partial z/\partial W_{h,z} \end{bmatrix}$$

$$\underline{\boldsymbol{\Delta}_1} = \left(\begin{bmatrix} \partial z/\partial W_{c,f} & \partial z/\partial W_{d,f} & \partial z/\partial W_{e,f} \\ \partial z/\partial W_{c,g} & \partial z/\partial W_{d,g} & \partial z/\partial W_{e,g} \\ \partial z/\partial W_{c,h} & \partial z/\partial W_{d,h} & \partial z/\partial W_{e,h} \end{bmatrix}\right)$$

$$\boldsymbol{\Delta}_0 = \ldots$$

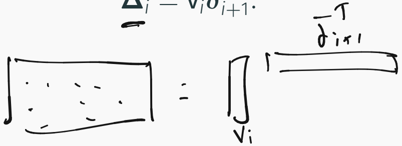
BACKPROP LINEAR ALGEBRA



$O(km)$

Claim 2: Weight derivative computation is an outer-product between the $(i+1)^{\text{st}}$ derivative vector and the i^{th} value vector.

$$\underline{\mathbf{\Delta}_i} = \mathbf{v}_i \bar{\boldsymbol{\delta}}_{i+1}^T.$$



What is the computational complexity of computing the derivatives for a single weight matrix $\mathbf{W}_i \in \mathbb{R}^{k\times m}$?

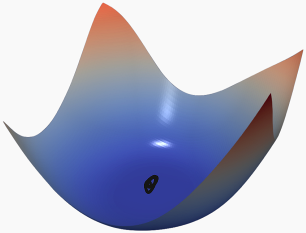
$O(p)$

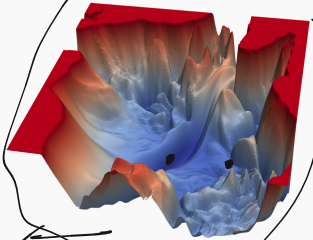
Takeaways: $O(d)$

- Backpropagation can be used to compute derivatives for all weights and biases for any feedforward neural network.
- Total computation cost is linear in the number of parameters of the network to compute $f(\boldsymbol{\theta}, \mathbf{x})$ and thus $\nabla L\,(y, f(\boldsymbol{\theta}, \mathbf{x}))$ for a single training example $\mathbf{x}, \mathbf{y}$.
- SGD can be run in $O(P)$ time per iteration for a network with P parameters.
- Final computation boils down to linear algebra operations (matrix multiplication and vector operations) which can be performed quickly on a GPU.

CONVERGENCE

Least squares regression, logistic regression, SVMs, even all of these with kernels lead to convex losses.


convex loss

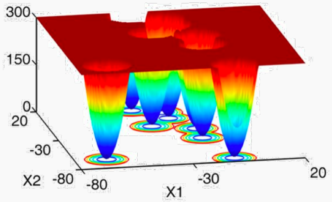

cross-entropy loss for neural net

Neural networks very much do not…

CONVERGENCE

But SGD still performs remarkably well in practice. Understanding this phenomenon is still an open research question in machine learning and optimization. Current hypotheses include:

- Initialization seems important (random uniform vs. random Gaussian vs. Xavier initialization vs. He initialization vs. etc.)
- Randomization helps in escaping local minima.
- Many local minima are global minima?
- SGD finds "good" local minima?



AUTODIFF

Issue: Backpropagation + SGD is fast, but tedious to implement.

Typical to use automatic differentiation, which can compute the gradient of pretty much any function you can code up.

```
def loss(W, b):
    preds = predict(W, b, inputs)
    label_probs = preds * targets + (1 - preds) * (1 - t
    return -np.sum(jnp.log(label_probs))

from jax import grad
W_grad, b_grad = grad(loss, (0, 1))(W, b)
```

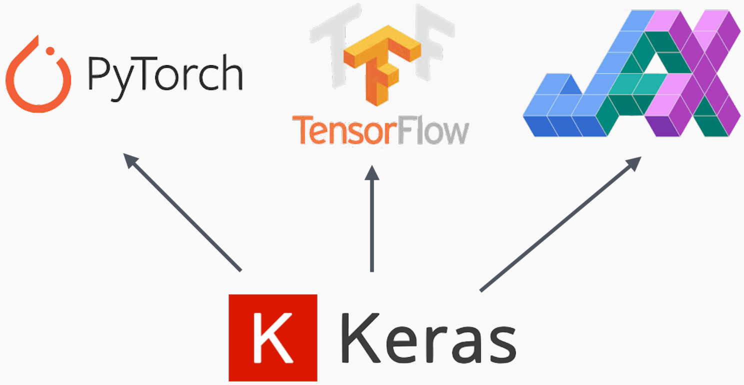
LIBRARIES

May mature low-level libraries that handle neural network representation, autodiff, have built in optimizers (SGD, ADAM, etc.), etc.

PyTorch TensorFlow

LIBRARIES

Higher-level libraries like Keras make it even easy to work with this software. Tools for easily defining and building neural networks with specific structure, tracking training, etc.



LIBRARIES

Define:

```
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

Compile:

Break until 3:27.

```
opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train:

```
hist = model.fit(Xtr, ytr, epochs=30, batch_size=100, validation_data=(Xts,yts))
```

Last week we released two demos on working with Keras:
`keras_demo_synthetic.ipynb` and
`keras_demo_mnist.ipynb`

CONVOLUTIONAL NETS

FEATURE EXTRACTION

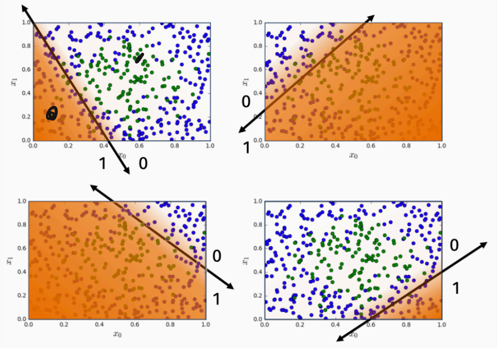
Why do neural networks work so well?

Treat feature transformation/extraction as part of the learning process instead of making this the users job.

But sometimes they still need a nudge in the right direction...

BASIC FEATURE EXTRACTION

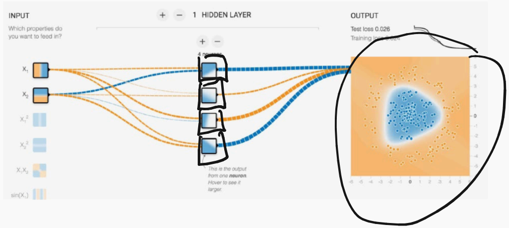
Sigmoid activation: Each hidden variable h_i equals $\frac{1}{1+e^{-\bar{h}_i}}$ where $\bar{h}_i = \mathbf{w}^T\mathbf{x} + b$ for input $\mathbf{x}$.



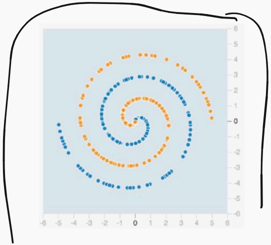
Other non-linearities yield similar features.

BASIC FEATURE EXTRACTION

If you combine more hidden variables, you can start building more complex classifiers.

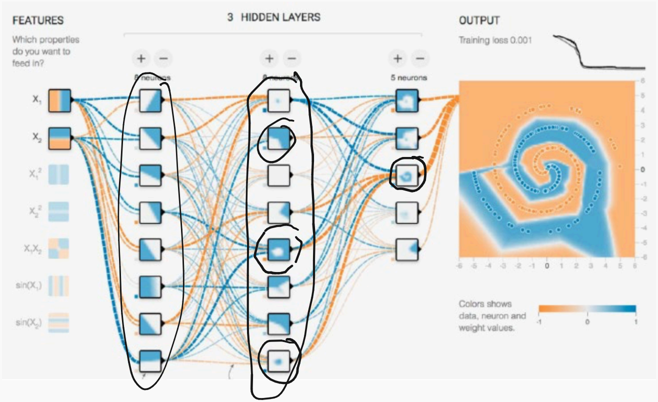


What about even more complex datasets?



BASIC FEATURE EXTRACTION

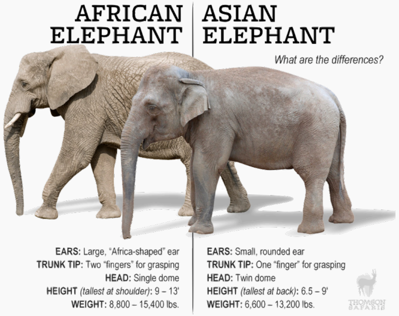
With more layers, complexity starts ramping up:



But there is a limit...

BASIC FEATURE EXTRACTION

Modern machine learning algorithms can differentiate between images of African and Asian elephants:


AFRICAN ELEPHANT
ASIAN ELEPHANT
What are the differences?
EARS: Large, "Africa-shaped" ear
TRUNK TIP: Two "fingers" for grasping
HEAD: Single dome
HEIGHT (tallest at shoulder): 9 – 13'
WEIGHT: 8,800 – 15,400 lbs.
EARS: Small, rounded ear
TRUNK TIP: One "finger" for grasping
HEAD: Twin dome
HEIGHT (tallest at back): 6.5 – 9'
WEIGHT: 6,600 – 13,200 lbs.
THOMSON SAFARIS

The features needed for this task are <u>far more complex</u> then we could expect a network to learn completely on its own using combinations of linear layers + non-linearities.

CONVOLUTIONAL FEATURE EXTRACTION

Remainder of lecture: Understand why convolution is a powerful way of extracting features from image data. Also super valuable for

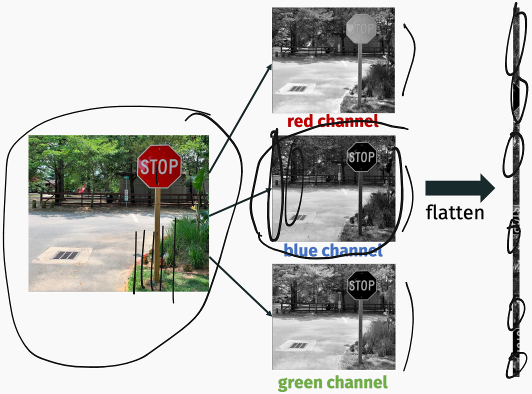
- (Audio data.)
- (Time series data.)

Ultimately, can build convolutional networks that already have convolutional feature extraction pre-coded in.

Just one way of "nudging" the neural network in the right direction. I.e., deciding on an architecture to match our specific data. Different data requires different "nudges".

MOTIVATING EXAMPLE

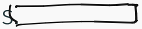
What features would tell use this image contains a stop sign?



Typically way of vectorizing an image chops up and splits up any pixels in the stop sign. We need very complex features to piece these back together again...

CONVOLUTION

Objects or features of an image often involve pixels that are spatially correlated. Convolution explicitly encodes this.



Definition (Discrete 1D convolution[1])

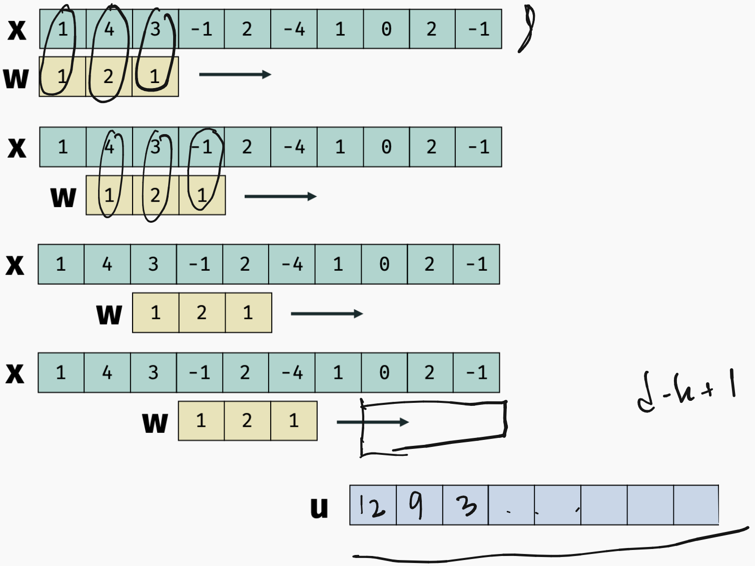
Given $\underline{\mathbf{x} \in \mathbb{R}^d}$ and $\underline{\mathbf{w} \in \mathbb{R}^k}$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $d - k + 1$ vector with:

$$[\mathbf{x} \circledast \mathbf{w}]_i = \sum_{j=1}^{k} \overset{i+j-1}{\mathbf{x}_{(j+i-1)}} \mathbf{w}_j$$

Think of $\mathbf{x} \in \mathbb{R}^d$ as long **data vector** (e.g. d = 512) and $\mathbf{w} \in \mathbb{R}^k$ as short **filter vector** (e.g. k = 8). $\mathbf{u} = [\mathbf{x} \circledast \mathbf{w}]$ is a feature transformation.

[1]This is slightly different from the definition of convolution you might have seen in a Digital Signal Processing class because **w** does not get “flipped”. In signal processing our operation would be called correlation.)

1D convolution



MATCH THE CONVOLUTION

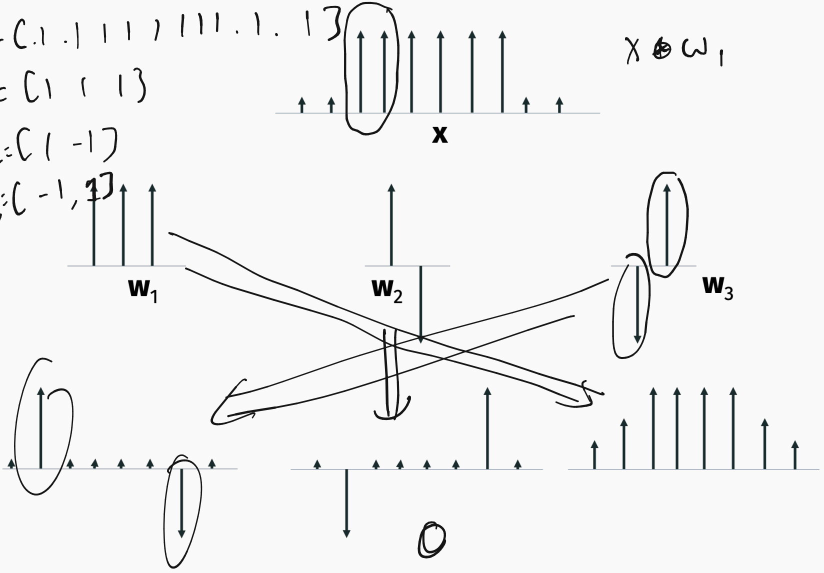
$x = [.1\ .1\ 1\ 1\ 1\ 1\ 1\ 1\ .1\ .1]$

$\omega_1 = [1\ 1\ 1]$

$\omega_2 = [1\ -1]$

$\omega_3 = [-1, 1]$

$x \circledast \omega_1$



2D convolution

Definition (Discrete 2D convolution)

Given matrices $(\mathbf{x} \in \mathbb{R}^{d_1 \times d_2})$ and $(\mathbf{w} \in \mathbb{R}^{k_1 \times k_2})$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $\underline{(d_1 - k_1 + 1)} \times (\underline{d_2 - k_2} + 1)$ matrix with:

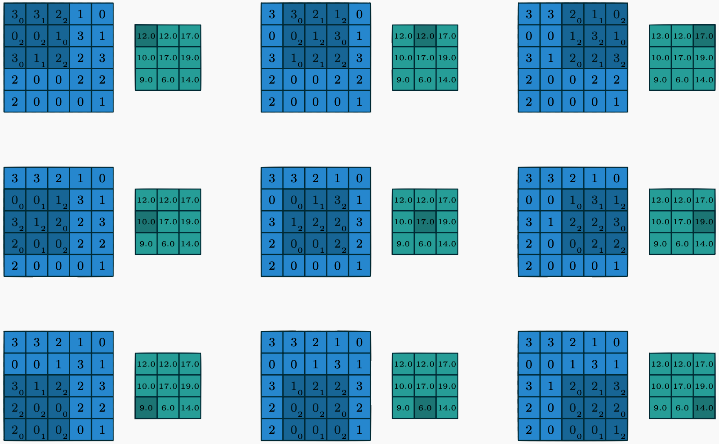
$$\left([\mathbf{x} \circledast \mathbf{w}]_{i,j} = \sum_{\ell=1}^{k_1} \sum_{h=1}^{k_2} \mathbf{x}_{(i+\ell-1),(j+h-1)} \cdot \mathbf{w}_{\ell,h} \right)$$

Again technically this is "correlation" not "convolution". Should be performed in Python using `scipy.signal.correlate2d` instead of `scipy.signal.convolve2d`.

$\mathbf{w}$ is called the filter or convolution kernel and again is typically much smaller than $\mathbf{x}$.

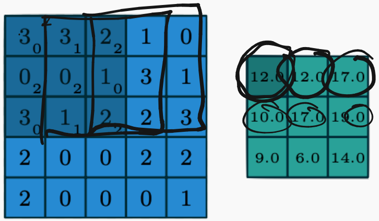
2D convolution

$$w = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$



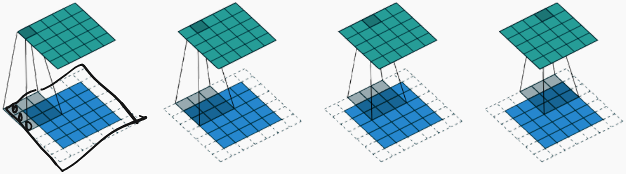
2D convolution

$$\mathrm{w} = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$



ZERO PADDING

Sometimes "zero-padding" is introduced so $\mathbf{x} \circledast \mathbf{w}$ is $\underline{d_1} \times \underline{d_2}$ if $\mathbf{x}$ is $d_1 \times d_2$.



Need to pad on left and right by $\underline{(k_1 - 1)/}2$ and on top and bottom by $\underline{(k_2 - 1)/}2$.

APPLICATIONS OF CONVOLUTION

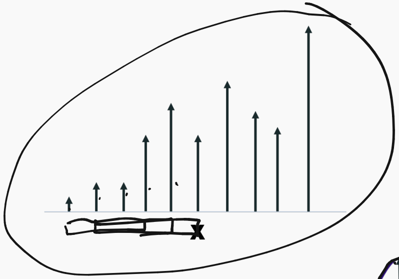
Examples code will be available in `demo1_convolutions.ipynb`.

Application 1: Blurring/smooth.

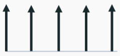
In one dimension:

- Uniform (moving average) filter: $w_i = \frac{1}{k}$ for $i = 1, \ldots, k$.
- Gaussian filter: $w_i \sim \exp^{(i-k/2)^2/\sigma^2}$ for $i = 1, \ldots, k$.

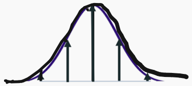
SMOOTHING FILTERS



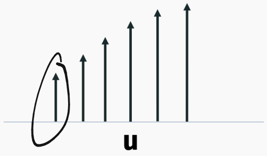
e^{-x^2}

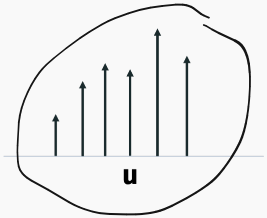


Uniform filter
(moving average)



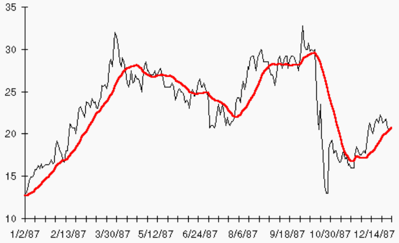
Gaussian filter





SMOOTHING FILTERS

Useful for smoothing time-series data, or removing noise/static from audio data.

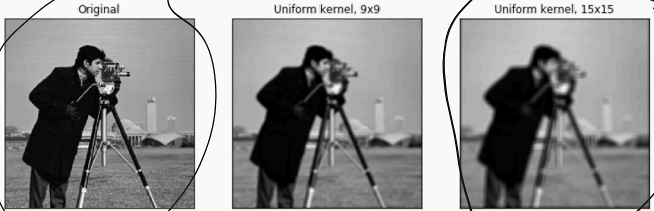


Replaces every data point with a local average.

SMOOTHING IN TWO DIMENSIONS

In two dimensions:

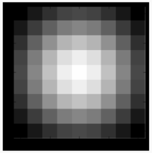
- Uniform filter: $\mathbf{w}_{i,j} = \frac{1}{k_1 k_2}$ for $i = 1, \ldots, k_1$, $j = 1, \ldots, k_2$.
- Gaussian filter: $\mathbf{w}_i \sim \exp^{\frac{(i-k_1/2)^2+(j-k_2/2)^2}{\sigma^2}}$ for $i = 1, \ldots, k_1$, $j = 1, \ldots, k_2$.

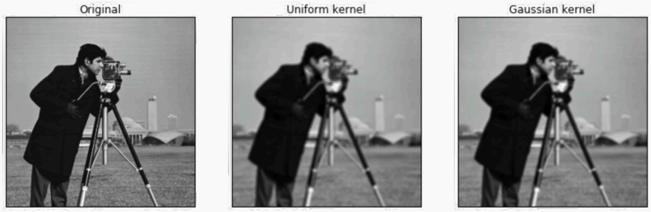


Larger filter equates to more smoothing.

SMOOTHING IN TWO DIMENSIONS

For Gaussian filter, you typically choose $k \gtrsim \underline{2\sigma}$ to capture the fall-off of the Gaussian.

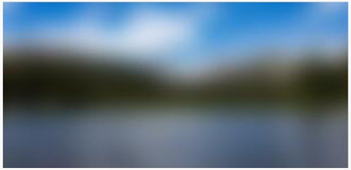
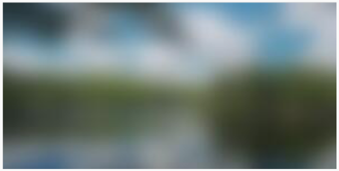




Both approaches effectively denoise and smooth images.

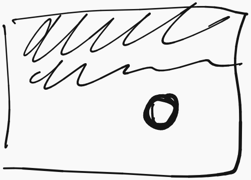
SMOOTHING FOR FEATURE EXTRACTION

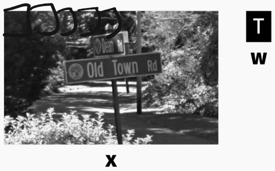
When combined with other feature extractors, smoothing at various levels allows the algorithm to focus on high-level features over low-level features.

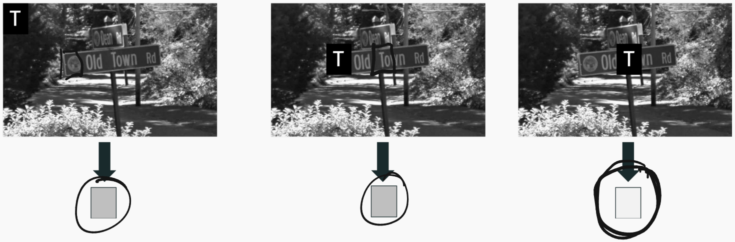



Application 2: Pattern matching.

Slide a pattern over an image. Output of convolution will be higher when pattern correlates well with underlying image.







LOCAL PATTERN MATCHING

Applications of local pattern matching:

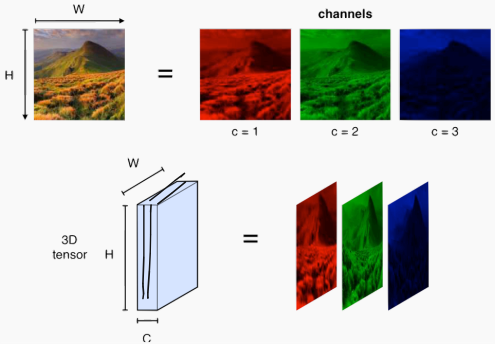
- Check if an image contains text.
- Look for specific sound in audio recording.
- Check for other well-structured objects

3D CONVOLUTION

Recall that color images actually have three color channels for **red**, **green**, **blues**. Each pixel is represented by 3 values (e.g. in $0, \ldots, 255$) giving the intensity in each channel.

$[0, 0, 0]$ = black, $[0, 0, 0]$ = white, $[1, 0, 0]$ = pure red, etc.

View image as 3D tensor:

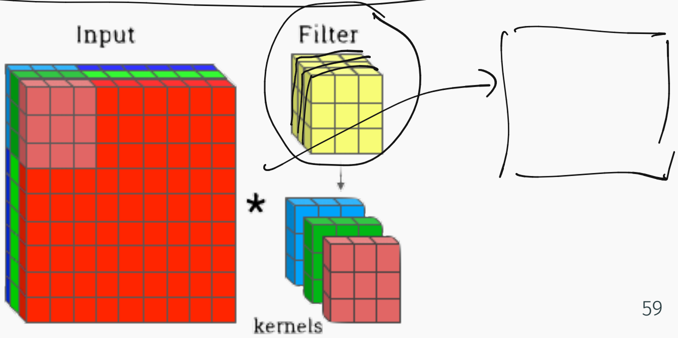


3D CONVOLUTION

Definition (Discrete 3D convolution)

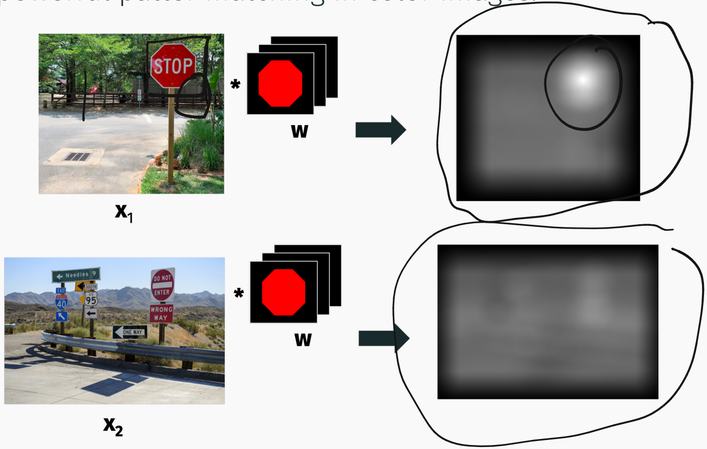
Given tensors $\mathbf{x} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ and $\mathbf{w} \in \mathbb{R}^{k_1 \times k_2 \times k_3}$ the discrete convolution $\mathbf{x} \circledast \mathbf{w}$ is a $(d_1 - k_1 + 1) \times (d_2 - k_2 + 1) \times (d_3 - k_3 + 1)$ tensor with:

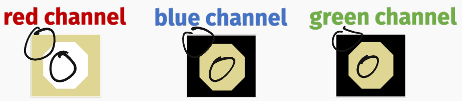
$$[\mathbf{x} \circledast \mathbf{w}]_{i,j,g} = \sum_{\ell=1}^{k_1} \sum_{m=1}^{k_2} \sum_{n=1}^{k_3} \mathbf{x}_{(i+\ell-1),(j+m-1),(g+n-1)} \cdot \mathbf{w}_{\ell,m,n}$$



APPLICATION 2: PATTERN MATCHING

More powerful patter matching in color images:





🟨	= -1
⬛	= 0
⬜	= 1

APPLICATIONS OF CONVOLUTION

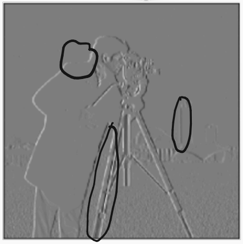
Application 3: Edge detection.

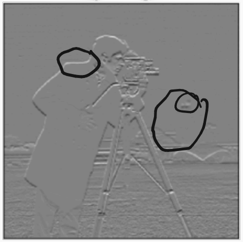
These are 2D edge detection filter:

vertical

horizontal

$$W_1 = \begin{bmatrix} 1 & -1 \end{bmatrix} \qquad W_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$


x * ?


x * ?

$x_{i,j}$

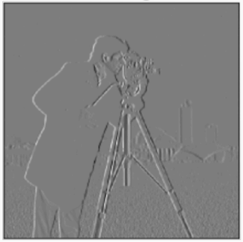
D

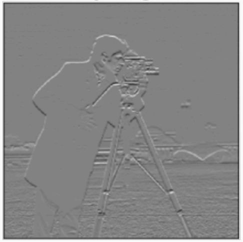
$x_{ij} - x_{i(j+1)}$

APPLICATIONS OF CONVOLUTION

Sobel filter is more commonly used:

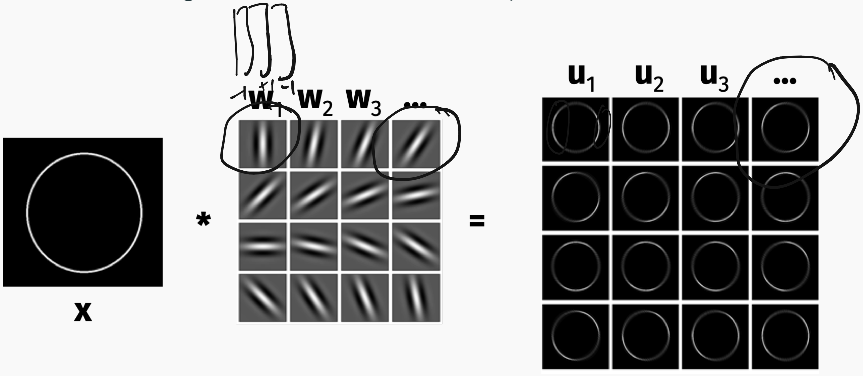
$$W_1 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad W_2 = \begin{array}{ccccc} 0 & 1 & 2 & 1 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ & 0 & 0 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ & & & & 0 \\ 0 & -1 & -2 & -1 & 0 \end{array}$$


x * ?


x * ?

DIRECTIONAL EDGE DETECTION

Can define edge detection filters for any orientation.

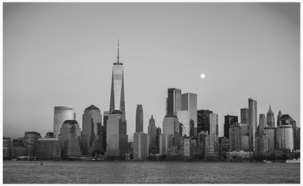


-1 +1 -1
$\mathbf{w}_1$ $\mathbf{w}_2$ $\mathbf{w}_3$ ...
$\mathbf{u}_1$ $\mathbf{u}_2$ $\mathbf{u}_3$ ...
$\mathbf{x}$
*
=

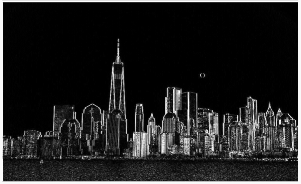
EDGE DETECTION

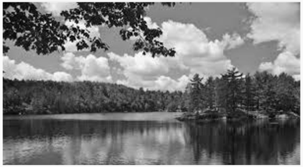
How would edge detection as a feature extractor help you classify images of city-scapes vs. images of landscapes?

EDGE DETECTION

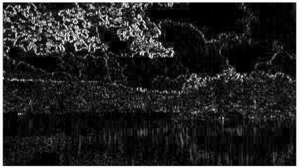

$\mathbf{I}_C$

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \longrightarrow$$


$\mathbf{E}_C$


$\mathbf{I}_L$

$$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \longrightarrow$$

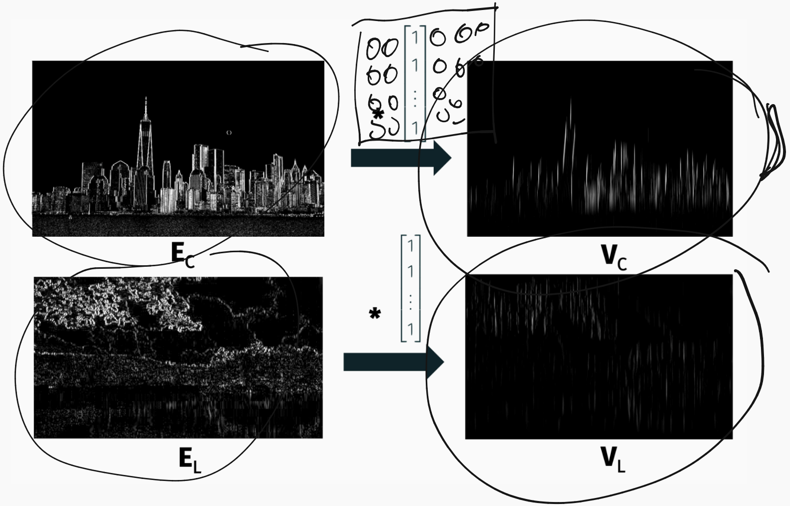

$\mathbf{E}_L$

$$\texttt{mean}(\mathrm{E}_C) = .108 \quad \text{vs.} \quad \texttt{mean}(\mathrm{E}_L) = .123$$

The image with highest vertical edge response isn't the city-scape.

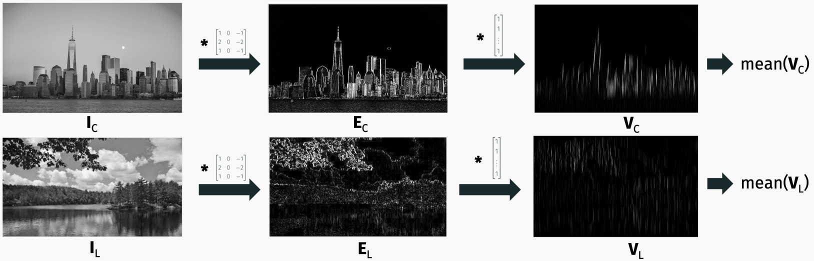
EDGE DETECTION + PATTERN MATCHING

Feed edge detection result into pattern matcher that looks for long vertical lines.



$\mathbf{E}_C$
$\mathbf{V}_C$
$\mathbf{E}_L$
$\mathbf{V}_L$
$* \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$
$* \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$

HIERARCHICAL CONVOLUTIONAL FEATURES



$\mathbf{I}_C$
$\mathbf{E}_C$
$\mathbf{V}_C$
mean($\mathbf{V}_C$)
$\mathbf{I}_L$
$\mathbf{E}_L$
$\mathbf{V}_L$
mean($\mathbf{V}_L$)
$* \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$
$* \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$

$$\texttt{mean}(\mathbf{V}_C) = .062 \quad \text{vs. } \texttt{mean}(\mathbf{V}_L) = .054$$

The image with highest average response to (edge detector) + (vertical pattern) is the city scape.

$\text{mean}(\mathbf{V}) = \mathbf{V}^T\boldsymbol{\beta}$ where $\boldsymbol{\beta} = [1/n, \ldots, 1/n]$. So the new features in $\mathbf{V}$ could be combined with a simple linear classifier to separate cityscapes from landscapes.

HIERARCHICAL CONVOLUTIONAL FEATURES

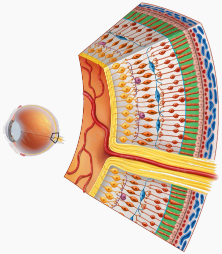
Hierarchical combinations of simple convolution filters are very powerful for understanding images.

Edge detection seems like a critical first step.

Lots of evidence from biology.

VISUAL SYSTEM

Light comes into the eye through the lens and is detected by an array of photosensitive cells in the **retina**.



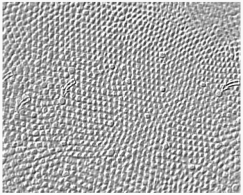
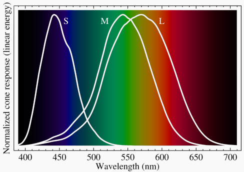


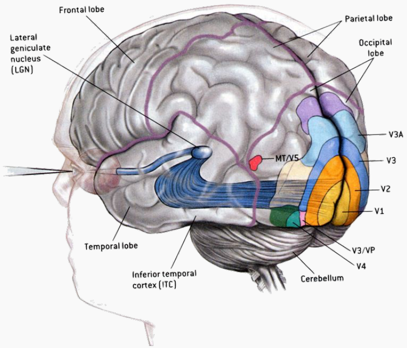
Fig. 13. Tangential section through the human fovea. Larger cones (arrows) are blue cones. From Ahnelt et al. 1987.

Rod cells are sensitive to all light, larger **cone** cells are sensitive to specific colors. We have three types of cones:



VISUAL SYSTEM

Signal passes from the retina to the primary (V1) visual cortex, which has neurons that connect to higher level parts of the brain.

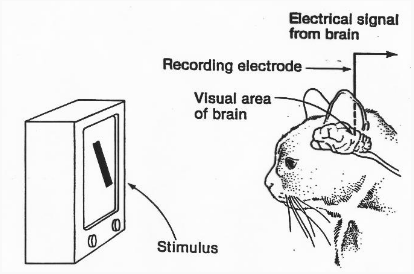


What sort of processing happens in the primary cortex?

Lots of edge detection!

EDGE DETECTORS IN CATS

Huber + Wiesel, 1959: “Receptive fields of single neurones in the cat’s striate cortex.” Won Nobel prize in 1981.



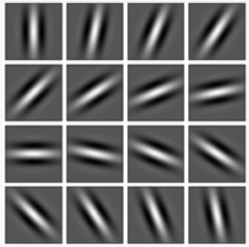
Different neurons fire when the cat is presented with stimuli at different angles. Cool video at `https://www.youtube.com/watch?v=OGxVfKJqX5E`.

”What the Frog’s Eye Tells the Frog’s Brain”, Lettvin et al. 1959. Found explicit edge detection circuits in a frogs visual cortex.

EXPLICIT FEATURE ENGINEERING

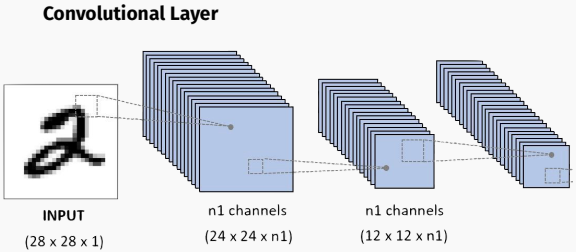
State of the art until 13 years ago:

- Convolve image with edge detection filters at many different angles.
- Hand engineer features based on the responses.
- SIFT and HOG features were especially popular.



CONVOLUTIONAL NEURAL NETWORKS

Neural network approach: Learn the parameters of the convolution filters based on training data.



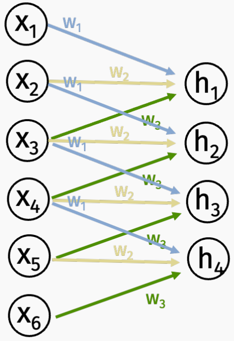
First convolutional layer involves n convolution filters $\mathbf{W}_1, \ldots, \mathbf{W}_q$. Each is small, e.g. 5×5. Every entry in $\mathbf{W}_i$ is a free parameter: $\sim 25 \cdot q$ parameters to learn.

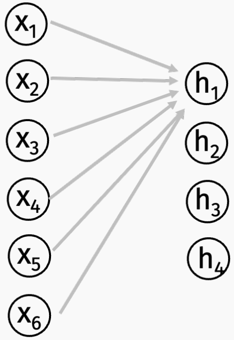
Produces q matrices of hidden variables: i.e. a tensor with depth q.

Each output in the tensor is processed with a **non-linearity**. Most commonly a Rectified Linear Unity (ReLU): $x = \max(\bar{x}, 0)$.

WEIGHT SHARING

Convolutional layers can be viewed as fully connected layers with added constraints. Many of the weights are forced to 0 and we have weight sharing constraints.



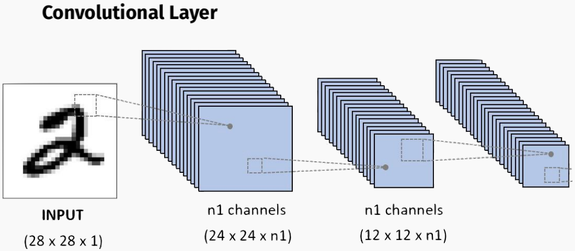


Weight sharing needs to be accounted for when running backprop/gradient descent.

CONVOLUTIONAL NEURAL NETWORKS

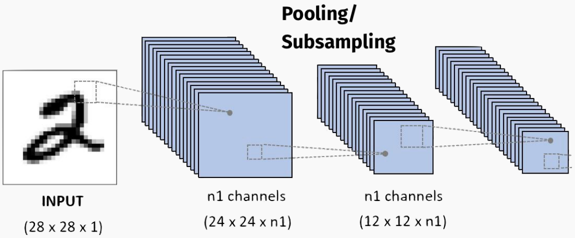
A fully connected layer that extracts the same features would require $(28 \cdot 28 \cdot 24 \cdot 24) \cdot q = 451{,}584 \cdot q$ parameters. Difference of over $200{,}000$x from $25q$.

By "baking in" knowledge about what type of features matter, we greatly simplify the network.



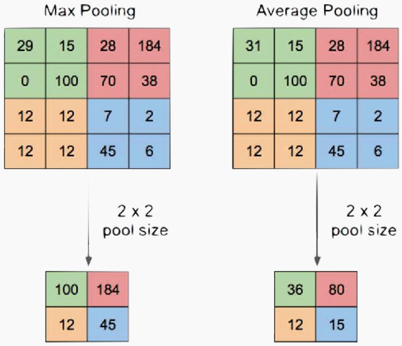
POOLING AND DOWNSAMPLING

Convolution + non-linearity are typically followed by a layer which performs **pooling + down-sampling**.

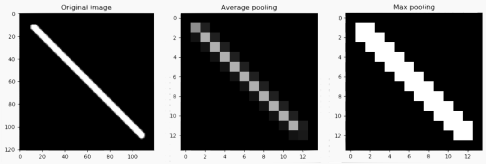


Most common approach is max-pooling.

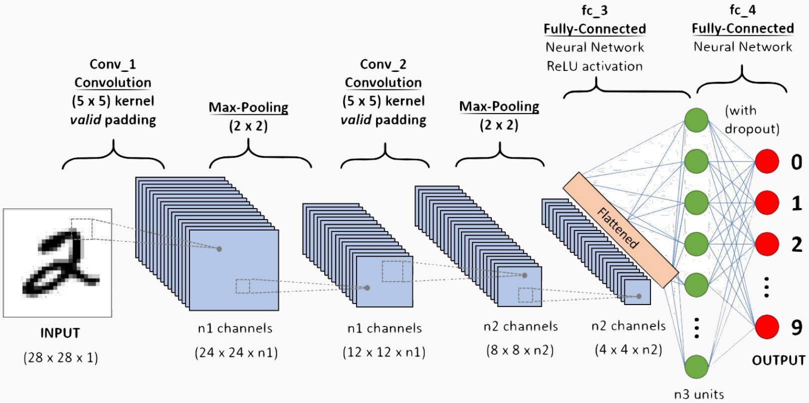
POOLING AND DOWNSAMPLING



- Reduces number of variables.
- Helps “smooth” result of convolutional filters.
- Improves shift-invariance.



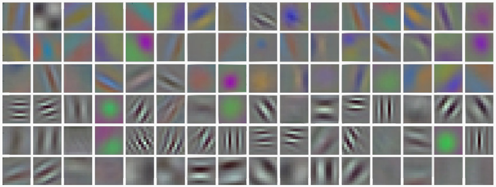
OVERALL NETWORK ARCHITECTURE



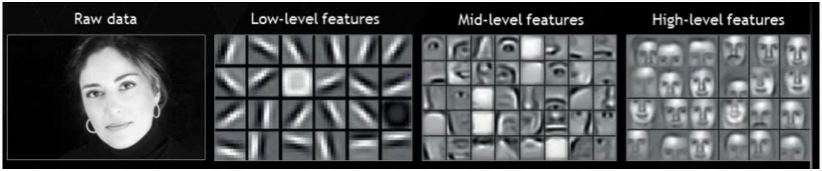
Each layer contains a 3D tensor of variables. Last few layers are standard fully connected layers.

UNDERSTANDING LAYERS

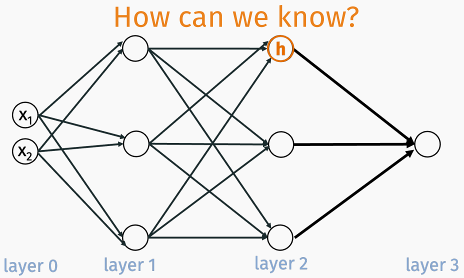
What type of convolutional filters do we learn from gradient descent?
Lots of edge detectors in the first layer!



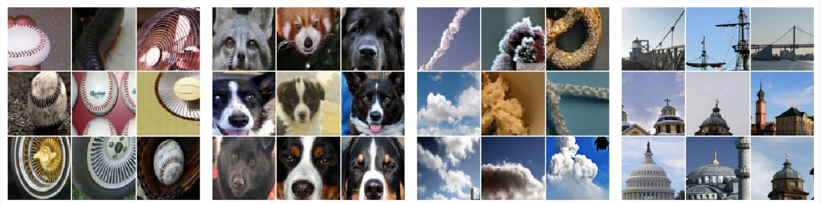
Other layers are harder to understand... but roughly hidden variables later in the network encode for “higher level features”:



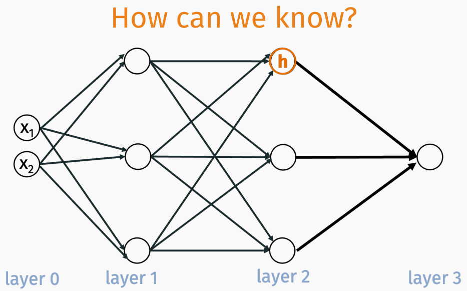
UNDERSTANDING LAYERS



Go through dataset and find the inputs that most "excite" a given neuron h. I.e. for which $|h(\mathbf{x})|$ is largest.



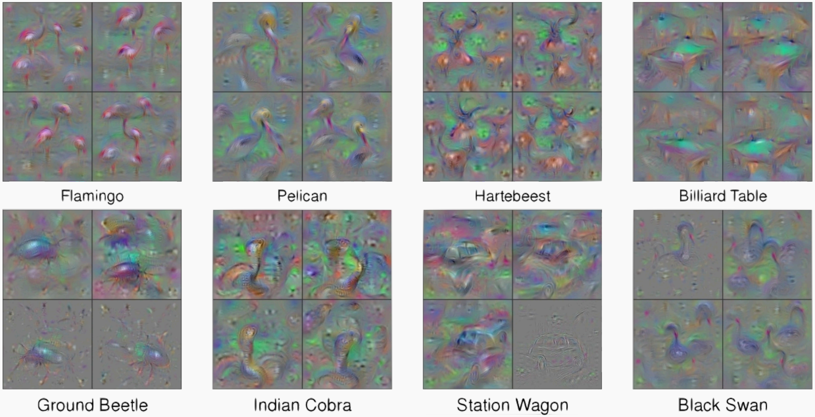
UNDERSTANDING LAYERS



Alternative approach: Solve the optimization problem $\max_{\mathbf{x}} |h(\mathbf{x})|$ e.g. using gradient descent.

UNDERSTANDING LAYERS

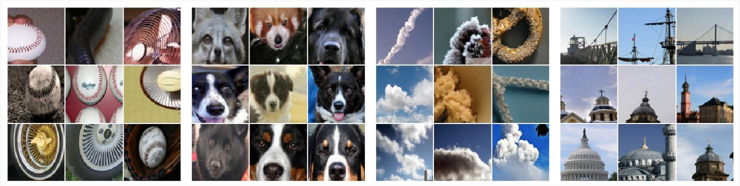
Early work had some interesting results.

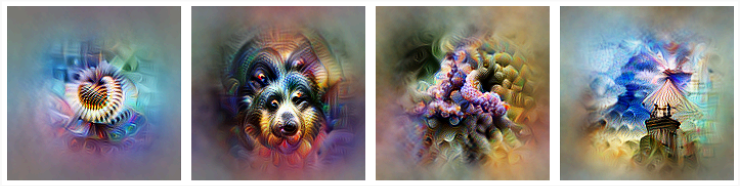


“Understanding Neural Networks Through Deep Visualization”, Yosinski et al.

UNDERSTANDING LAYERS

There has been a lot of work on improving these methods by regularization. I.e. solve $\max_{\mathbf{x}} |h(\mathbf{x})| + g(\mathbf{x})$ where g constrains $\mathbf{x}$ to look more like a “natural image”.

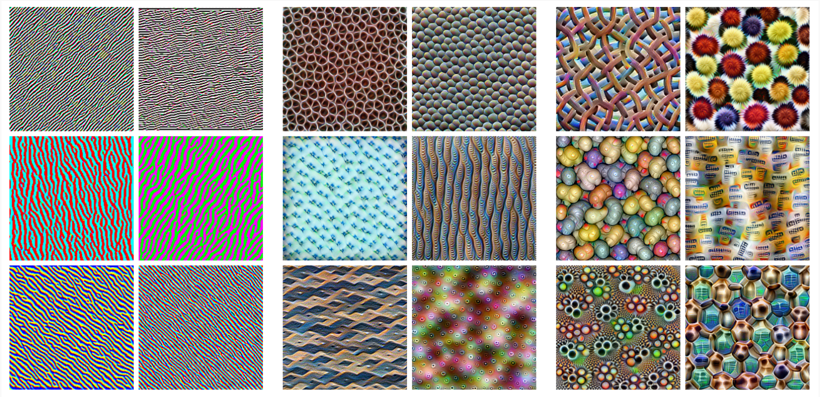




If you are interested in learning more on these techniques, there is a great Distill article at: `https://distill.pub/2017/feature-visualization/`.

UNDERSTANDING LAYERS

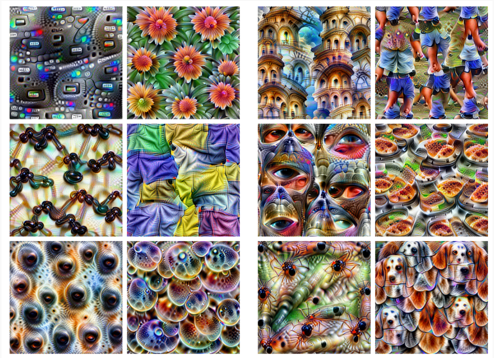
Nodes at different layers have different layers capture increasingly more abstract concepts.



Edges (layer conv2d0) **Textures** (layer mixed3a) **Patterns** (layer mixed4a)

UNDERSTANDING LAYERS

Nodes at different layers have different layers capture increasingly more abstract concepts.


Parts (layers mixed4b & mixed4c) **Objects** (layers mixed4d & mixed4e)

General obervation: Depth more important than width. Alexnet 2012 had 8 layers, modern convolutional nets can have 100s.

TRICKS OF THE TRADE

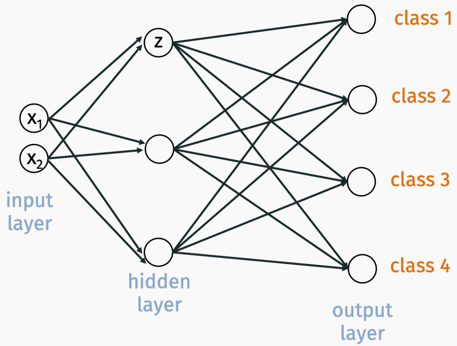
Beyond techinques discussed for general neural nets (back-prop, batch gradient descent, adaptive learning rates) training deep networks requires a lot of “tricks”.

- Batch normalization (accelerate training).
- Dropout (prevent over-fitting)
- Residual connections (accelerate training, allow for more depth – 100s of layers).
- Data augmentation.

And deep networks require lots of training data and lots of time.

BATCH NORMALIZATION

Start with any neural network architecture:



For input $\mathbf{x}$,

$$\bar{z} = \mathbf{w}^T\mathbf{x} + b$$
$$z = s(\bar{z})$$

where $\mathbf{w}$, b, and s are weights, bias, and non-linearity.

BATCH NORMALIZATION

$\bar{z}$ is a function of the input $\mathbf{x}$. We can write it as $\bar{z}(\mathbf{x})$. Consider the mean and standard deviation of the hidden variable over our entire dataset $\mathbf{x}_1 \ldots, \mathbf{x}_n$:

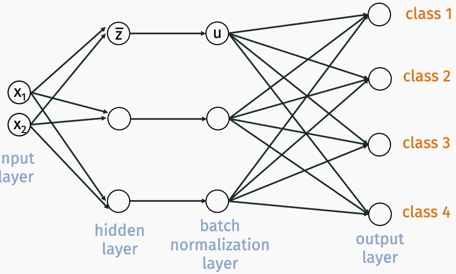
$$\mu = \frac{1}{n}\sum_{j=1}^{n} \bar{z}(\mathbf{x}_j)$$

$$\sigma^2 = \frac{1}{n}\sum_{j=1}^{n} (\bar{z}(\mathbf{x}_j) - \mu)^2$$

Just as normalization (mean centering, scaling to unit variance) is sometimes used for input features, batch-norm applies normalization to learned features.

BATCH NORMALIZATION

Can add a batch normalization layer after any layer:



$$\bar{u} = \frac{\bar{z} - \mu}{\sigma}$$
$$u = s(\bar{u}).$$

Has the effect of mean-centering/normalizing $\bar{z}$. Typically we actualy allow $u = s(\gamma \cdot \bar{u} + c)$ for learned parameters γ and c.

BATCH NORMALIZATION

Proposed in 2015: “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, Ioffe, Szegedy.

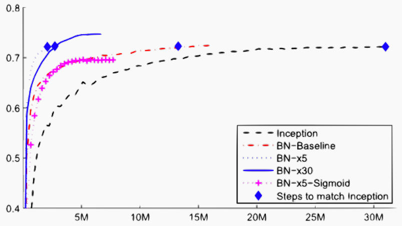


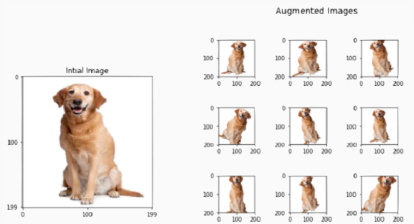
Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

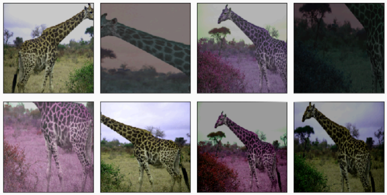
Doesn’t change the expressive power of the network, but allows for significant convergence acceleration. It is not yet well understood why batch normalizition speeds up training.

DATA AUGMENTATION

Great general tool to know about. **Main idea:**

- More training data typically leads to a more accurate model.
- Artificially enlarge training data with simple transformations.





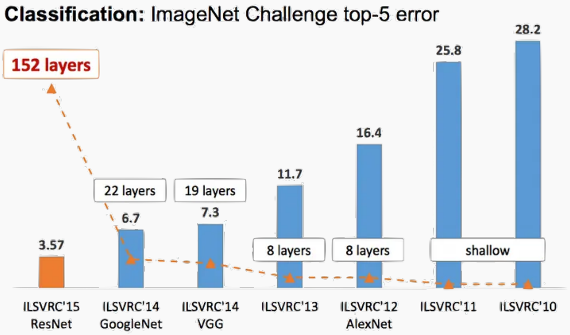
Take training images and randomly shift, flip, rotate, skew, darken, lighten, shift colors, etc. to create new training images. **Final classifier will be more robust to these transformations.**

DEEP LEARNING TRICKS

Need to take a full course on neural networks/deep learning to learn more! State-of-the-art techniques are constantly evolving.

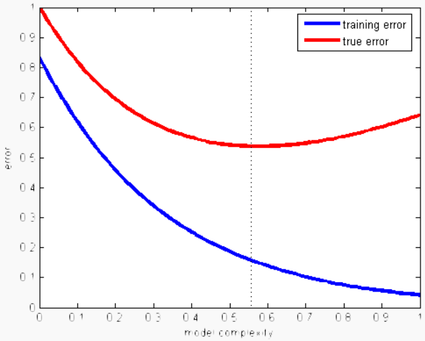
DEEPER AND DEEPER, BIGGER AND BIGGER

After AlexNet (8 layers, 60 million parameters) achieved start of the art performance on ImageNet, progress proceeded rapidly:



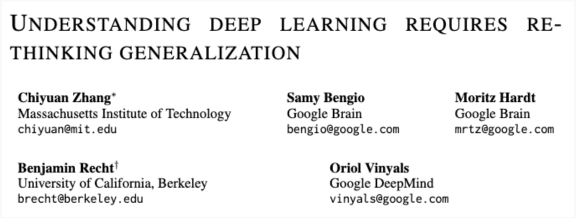
GENERALIZATION FOR NEURAL NETWORKS

Even with weight sharing, convolution, etc. modern neural networks typically have 100s of millions or billions of parameters. And we don't train them with regularization. Intuitively we might expect them to overfit to training data.



GENERALIZATION FOR NEURAL NETWORKS

In fact, we now know that modern neural nets can easily overfit to training data. This work showed that we can fit large vision data sets with random class labels to essentially perfect accuracy.


UNDERSTANDING DEEP LEARNING REQUIRES RE-THINKING GENERALIZATION

Chiyuan Zhang[*]
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

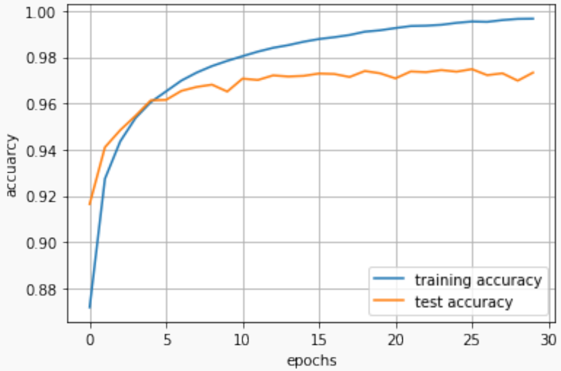
Benjamin Recht[†]
University of California, Berkeley
brecht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

But we don't always see a large gap between training and test error. **Don't take this to mean overfitting isn't a problem when using neural nets!** It's just not always a problem.

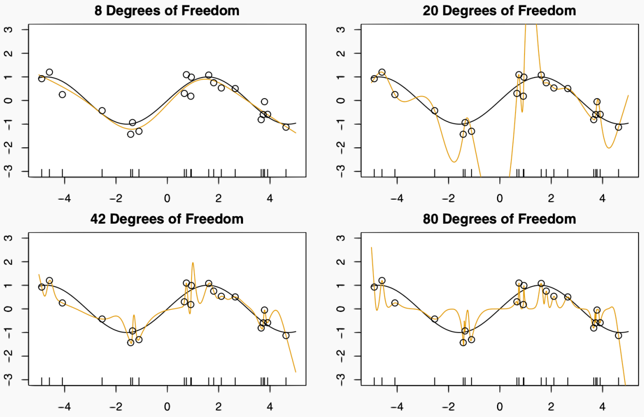
GENERALIZATION FOR NEURAL NETWORKS

We even see this lack of overfitting for MNIST data. See `keras_demo_mnist.ipynb` that I posted on the website:



GENERALIZATION FOR NEURAL NETWORKS

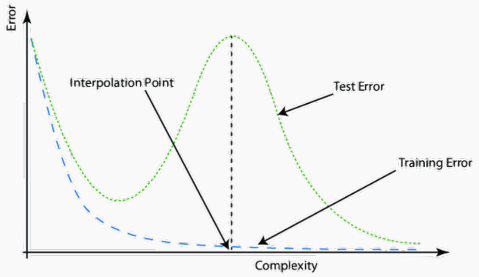
One growing realization is that this phenomena doesn't only apply to neural networks – it can also be true for fitting highly-overparameterized polynomials.



The choice of training algo (e.g. gradient descent) seems important.

DOUBLE DESCENT

We sometimes see a "double descent curve" for these models. Test error is worst for "just barely" overparameterized models, but gets better with lots of overparameterization.

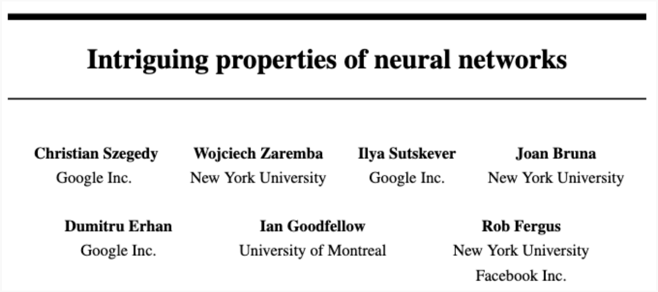


We don't usually see this same curve for neural networks.

OVERFITTING IN NEURAL NETS

Take away: Modern neural network overfit, but still seem fairly robust. Perform well on any new test data we throw that them.

Or do they?



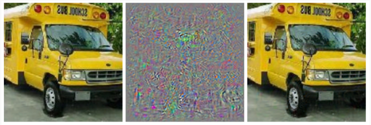
Intriguing properties of neural networks

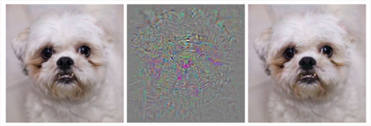
Christian Szegedy Google Inc.
Wojciech Zaremba New York University
Ilya Sutskever Google Inc.
Joan Bruna New York University
Dumitru Erhan Google Inc.
Ian Goodfellow University of Montreal
Rob Fergus New York University Facebook Inc.

ADVERSARIAL EXAMPLES

ADVERSARIAL EXAMPLES

Main discovery: It is possible to find imperceptibly small perturbations of input images that will fool deep neural networks. This seems to be a universal phenomenon.





Important: Random perturbations do not work!

ADVERSARIAL EXAMPLES

How to find "good" perturbations:

Fix model $f_{\boldsymbol{\theta}}$, input x, correct label y. Consider the loss $\ell(\theta, \mathsf{x}, y)$.

Solve the optimization problem:

$$\max_{\boldsymbol{\delta}, \|\boldsymbol{\delta}\| \leq \epsilon} \ell(\theta, \mathsf{x} + \boldsymbol{\delta}, y)$$

Can be solved using gradient descent! We just need to compute the derivative of the loss with respect to the image pixels. Backprop can do this easily.

ADVERSARIAL EXAMPLES

We will post a lab where you can find your own adversarial examples for a model called Resnet18. The entire model + weights are available pretrained through PyTorch, so we do not need to train it ourselves.