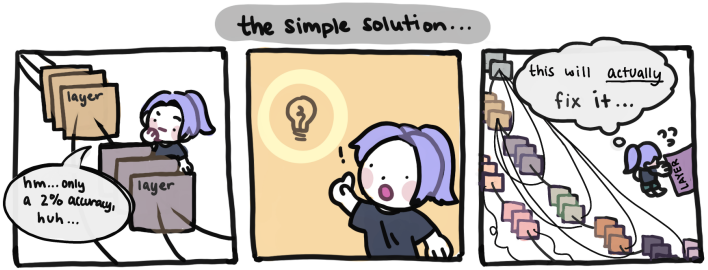
# Deep networks

CS 446 / ECE 449
2022-02-16 19:35:44 -0600 (0670b06)



`pytorch` **meta-algorithm.**

1. Clean/augment data (lecture 10?).
2. Pick model/architecture (anything from lectures 2-13).
3. Pick a loss function measuring model fit to data.
4. Run a gradient descent variant to fit model to data.
5. Tweak 1-4 until training error is small.
6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

- **Deep networks** have changed how much of machine learning is done, and vastly extended the use of machine learning.
- We'll re-visit the meta-algorithm later to see how things fit together.
- **Further reading:** UIUC faculty Svetlana Lazebnik has an entire course with excellent slides on deep learning.

# Plan for today

1. Basic deep network definition.
2. Convolutional layers.
3. Other standard layers.

**Plan for next few lectures:**

- Lecture 9: pytorch tutorial (will be updated before Tuesday) [jupyter notebook link] [pdf link]
- Lecture 10: continues with current slides.

# Basic network structure: iterated linear predictors

The most basic view of a neural network is an iterated linear predictor.

- 1 layer:
$$\boldsymbol{x} \mapsto \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1.$$
- 2 layers:
$$\boldsymbol{x} \mapsto \boldsymbol{W}_2 \left( \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1 \right) + \boldsymbol{b}_2.$$
- 3 layers:
$$\boldsymbol{x} \mapsto \boldsymbol{W}_3 \left( \boldsymbol{W}_2 \left( \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1 \right) + \boldsymbol{b}_2 \right) + \boldsymbol{b}_3.$$
- $L$ layers:
$$\boldsymbol{x} \mapsto \boldsymbol{W}_L \left( \cdots \left( \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1 \right) \cdots \right) + \boldsymbol{b}_L.$$

Alternatively, this is a composition of linear predictors:

$$\boldsymbol{x} \mapsto \left( f_L \circ f_{L-1} \circ \cdots \circ f_1 \right) (\boldsymbol{x}),$$

where $f_i(\boldsymbol{z}) = \boldsymbol{W}_i \boldsymbol{z} + \boldsymbol{b}_i$ is an affine function.

**Note:** "layer" terminology is ambiguous, we'll revisit it.

**Is there an issue with this slide...?**

# Wait a minute. . .

Note that

$$\begin{aligned} & \boldsymbol{W}_L \left( \cdots (\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) \cdots \right) + \boldsymbol{b}_L \\ = \; & (\boldsymbol{W}_L \cdots \boldsymbol{W}_1)\, \boldsymbol{x} + (\boldsymbol{b}_L + \boldsymbol{W}_L \boldsymbol{b}_{L-1} + \cdots + \boldsymbol{W}_L \cdots \boldsymbol{W}_2 \boldsymbol{b}_1) \\ = \; & \boldsymbol{w}^\mathsf{T} \left[ \begin{smallmatrix} \boldsymbol{x} \\ 1 \end{smallmatrix} \right], \end{aligned}$$

where $\boldsymbol{w} \in \mathbb{R}^{d+1}$ is

$$\boldsymbol{w}_{1:d}^\mathsf{T} = \boldsymbol{W}_L \cdots \boldsymbol{W}_1, \qquad \boldsymbol{w}_{d+1} = \boldsymbol{b}_L + \boldsymbol{W}_L \boldsymbol{b}_{L-1} + \cdots + \boldsymbol{W}_L \cdots \boldsymbol{W}_2 \boldsymbol{b}_1.$$

Oops, this is just a linear predictor.

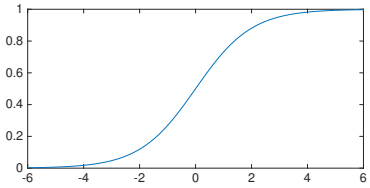
1. Activations/nonlinearities.

# Iterated logistic regression

Recall that logistic regression could be interpreted as a probability model:

$$\Pr[Y = 1|\boldsymbol{X} = \boldsymbol{x}] = \frac{1}{1 + \exp(-\boldsymbol{w}^\mathsf{T}\boldsymbol{x})} =: \sigma_\mathsf{s}(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}),$$

where $\sigma_\mathsf{s}$ is the logistic or sigmoid function

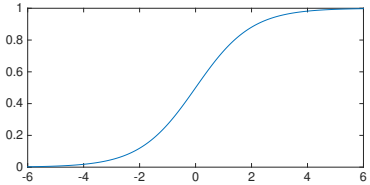


# Iterated logistic regression

Recall that logistic regression could be interpreted as a probability model:

$$\Pr[Y = 1 | \boldsymbol{X} = \boldsymbol{x}] = \frac{1}{1 + \exp(-\boldsymbol{w}^\mathsf{T} \boldsymbol{x})} =: \sigma_\mathsf{s}(\boldsymbol{w}^\mathsf{T} \boldsymbol{x}),$$

where $\sigma_\mathsf{s}$ is the logistic or sigmoid function




Now suppose $\sigma_\mathsf{s}$ is applied coordinate-wise, and consider

$$\boldsymbol{x} \mapsto (f_L \circ \cdots \circ f_1)(\boldsymbol{x}) \quad \text{where } f_i(\boldsymbol{z}) = \sigma_\mathsf{s}(\boldsymbol{W}_i \boldsymbol{z} + \boldsymbol{b}_i).$$

Don't worry, we'll slow down next slide;
for now, iterated logistic regression gave our first deep network!

Remark: can view intermediate layers as features to subsequent layers.

# Basic deep networks

A self-contained expression is

$$\boldsymbol{x} \mapsto \sigma_L \left( \boldsymbol{W}_L \sigma_{L-1} \left( \cdots \left( \boldsymbol{W}_2 \sigma_1 (\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \right) \cdots \right) + \boldsymbol{b}_L \right),$$

with equivalent "functional form"

$$\boldsymbol{x} \mapsto (f_L \circ \cdots \circ f_1)(\boldsymbol{x}) \quad \text{where } f_i(\boldsymbol{z}) = \sigma_i \left( \boldsymbol{W}_i \boldsymbol{z} + \boldsymbol{b}_i \right).$$

Some further details (many more to come!):

- $(\boldsymbol{W}_i)_{i=1}^L$ with $\boldsymbol{W}_i \in \mathbb{R}^{d_i \times d_{i-1}}$ are the weights, and $(\boldsymbol{b}_i)_{i=1}^L$ are the biases. Collectively, they are often simply called parameters.
- $(\sigma_i)_{i=1}^L$ with $\sigma_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_i}$ are called nonlinearties, or activations, or transfer functions, or link functions.
- This is only the basic setup.

# Choices of activation

Basic form:

$$\boldsymbol{x} \mapsto \sigma_L \left( \boldsymbol{W}_L \sigma_{L-1} \left( \cdots \boldsymbol{W}_2 \sigma_1 (\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \cdots \right) + \boldsymbol{b}_L \right).$$

Choices of activation (univariate, coordinate-wise):

- Indicator/step/heavyside/threshold $z \mapsto \mathbb{1}[z \geq 0]$.
  This was the original choice (1940s!).
- Sigmoid $\sigma_{\mathrm{s}}(z) := \frac{1}{1+\exp(-z)}$.
  This was popular roughly 1970s - 2005?
- Hyperbolic tangent $z \mapsto \tanh(z)$.
  Similar to sigmoid, used during same interval.
- Rectified Linear Unit (ReLU) $\sigma_{\mathrm{r}}(z) = \max\{0, z\}$.
  It (and slight variants, e.g., Leaky ReLU, ELU, ...) are the dominant choice now; popularized in "Imagenet/AlexNet" paper (Krizhevsky-Sutskever-Hinton, 2012).
  Original heavy use Fukushima (1960s).
- Identity $z \mapsto z$; we'll often use this as the last layer when we use cross-entropy loss.
- NON-coordinate-wise choices: we'll discuss "softmax" and "pooling".

# “Architectures” and “models”

Basic form:

$$\boldsymbol{x} \mapsto \sigma_L \left( \boldsymbol{W}_L \sigma_{L-1} \left( \cdots \boldsymbol{W}_2 \sigma_1 (\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 \cdots \right) + \boldsymbol{b}_L \right).$$

$((\boldsymbol{W}_i, \boldsymbol{b}_i))_{i=1}^L$, the weights and biases, are the parameters.
Let's roll them into $\mathcal{W} := ((\boldsymbol{W}_i, \boldsymbol{b}_i))_{i=1}^L$,
and consider the network as a two-parameter function $F_{\mathcal{W}}(\boldsymbol{x}) = F(\boldsymbol{x}; \mathcal{W})$.

- The model or class of functions is $\{F_{\mathcal{W}} : \text{all possible } \mathcal{W}\}$.
  $F$ (both arguments unset) is also called an architecture.
- To fit/train/optimize, typically we fix $F$ and vary $\mathcal{W}$.

# Affine expansion

We have been writing

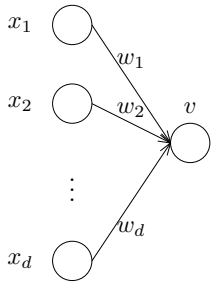
$$\boldsymbol{x} \mapsto \sigma_L \left( \cdots \left( \boldsymbol{W}_2 \sigma_1 \left( \boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1 \right) + \boldsymbol{b}_2 \right) \cdots \right),$$

rather than

$$\boldsymbol{x} \mapsto \sigma_L \left( \cdots \left( \boldsymbol{W}_2 \sigma_1 \left( \boldsymbol{W}_1 \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix} \right) \right) \cdots \right).$$

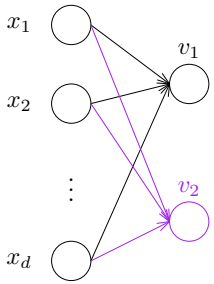
- First form seems natural:
  With "iterated linear prediction" perspective,
  it is natural to append 1 at every layer.
- Second form is sufficient:
  with ReLU, $\sigma_{\mathrm{r}}(1) = 1$, so can pass forward the constant;
  similar (but more complicated) options exist for other activations.
- Why do we do it?
  Vague belief that it helps...

# Classical network/graph perspective



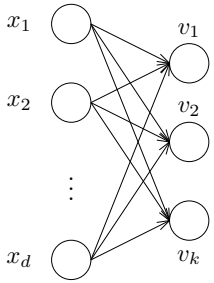

$$v := \sigma(z), \qquad z = \sum_{i=1}^{d} w_i x_i.$$

# Classical network/graph perspective



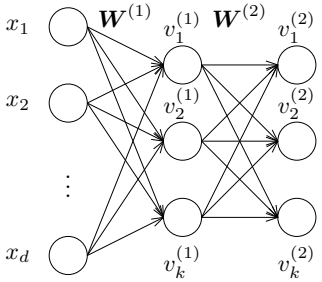

$$v_j := \sigma(z_j), \qquad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \qquad j \in \{1, 2\}.$$

# Classical network/graph perspective




$$v_j := \sigma(z_j), \qquad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \qquad j \in \{1, \dots, k\}.$$

# Classical network/graph perspective




$$v_j := \sigma(z_j), \qquad z_j := \sum_{i=1}^{d} W_{i,j} x_i, \qquad .$$

Classically, non-input/non-input nodes $(v_1^{(1)}, \ldots, v_k^{(1)})$ are a hidden layer.

# General graph-based view

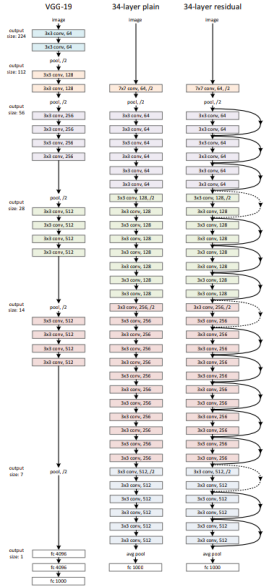
Classical graph-based perspective.

- Network is a directed acyclic graph;
  sources are inputs, sinks are outputs, intermediate nodes compute $\boldsymbol{z} \mapsto \sigma(\boldsymbol{w}^\mathsf{T}\boldsymbol{z} + b)$ (with their own $(\sigma, \boldsymbol{w}, b)$).
- Nodes at distance 1 from inputs are the first layer, distance 2 is second layer, and so on.

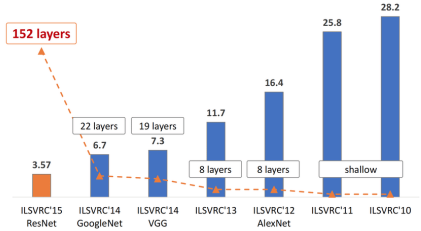
Current "computation graph" perspective.

- Edges can pass full tensors,
  which helps with current parallel hardware.
- Nodes are more general primitives,
  not only in the form $\sigma(\boldsymbol{W}\boldsymbol{z} + \boldsymbol{b})$.
- Edges will often "skip" layers;
  "layer" is therefore ambiguous.
- Diagram conventions differ;
  e.g., tensorflow graphs include nodes for parameters.

# Current-day networks: many layers...

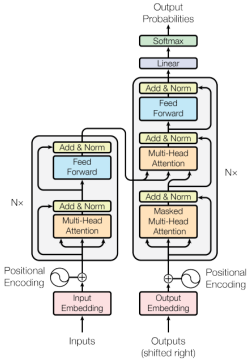



Taken from ResNet paper, 2015.

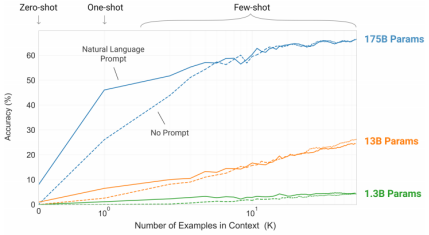



Taken from Nguyen et al, 2017.

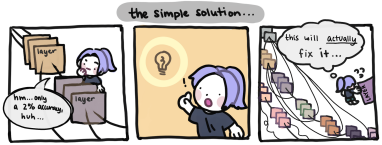
# . . . Many parameters. . .




Taken from "attention is all you need".
(Transformer paper,
Vaswani et al., 2017.)




Taken from GPT-3 paper.
(Brown et al., 2020.)



# Which architecture?

How do choose an architecture?

# Which architecture?

How do choose an architecture?

- How to choose $C$ and kernel in SVM?

# Which architecture?

How do choose an architecture?

- How to choose $C$ and kernel in SVM?
- Split data into training and validation,
  train different architectures and evaluate them on validation,
  choose architecture with lowest validiation error.
- As with other methods, this is a proxy to minimizing test error.

# Which architecture?

How do choose an architecture?

- How to choose $C$ and kernel in SVM?
- Split data into training and validation,
  train different architectures and evaluate them on validation,
  choose architecture with lowest validiation error.
- As with other methods, this is a proxy to minimizing test error.

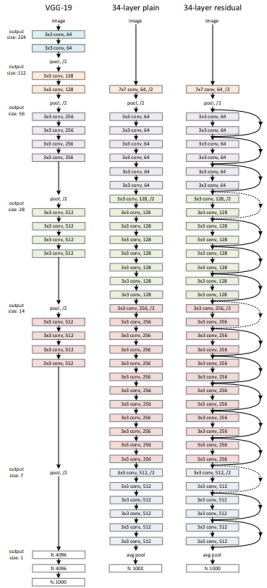
Note.

- For many standard tasks (e.g., classification of standard vision datasets), people have some preferred architectures.

  Even these have many attributes which may or may not help.
- For new problems and new domains, often there is extensive experimentation with architecture.

# Convolutional layers

# Convolutional layers




Taken from ResNet paper. 2015.

We're missing a few:

- Convolutions.
- Pooling.
- Skip connections.
- Batch norm.

# Continuous convolution in mathematics

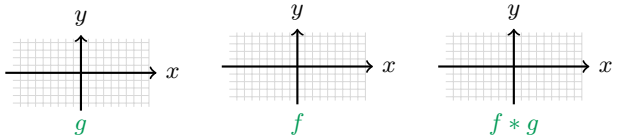
- Convolutions are typically continuous:

$$(f * g)(x) := \int f(y)g(x-y)\,\mathrm{d}y.$$

- Often, $f$ is 0 or tiny outside some small interval;
  e.g., if, $f$ is 0 outside $[-1, +1]$, then

$$(f * g)(x) = \int_{-1}^{+1} f(y)g(x-y)\,\mathrm{d}y.$$

  Think of this as sliding $f$, a filter, along $g$.



# Discrete convolutions in mathematics

We can also consider discrete convolutions:

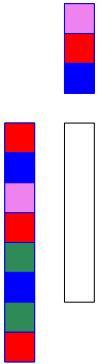
$$(f * g)(n) = \sum_{i=-\infty}^{\infty} f(i)g(n-i)$$

If both $f$ and $g$ are 0 outside some interval,
we can write this as matrix multiplication:

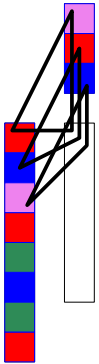
$$\begin{bmatrix} f(1) & 0 & \cdots & & \\ f(2) & f(1) & 0 & \cdots & \\ f(3) & f(2) & f(1) & 0 & \cdots \\ \vdots & & & & \\ f(d) & f(d-1) & f(d-2) & \cdots & \\ 0 & f(d) & f(d-1) & \cdots & \\ 0 & 0 & f(d) & \cdots & \\ \vdots & & & & \end{bmatrix} \begin{bmatrix} g(1) \\ g(2) \\ g(3) \\ \vdots \\ g(m) \end{bmatrix}$$

(The matrix at left is a "Toeplitz matrix".)
Note that we have padded with zeros;
the two forms are identical if $g$ starts and ends with $d$ zeros.

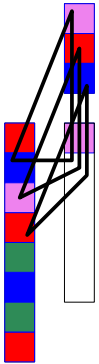
# 1-D convolution in deep networks



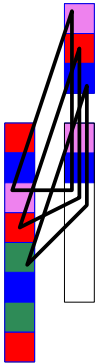
# 1-D convolution in deep networks



# 1-D convolution in deep networks



# 1-D convolution in deep networks

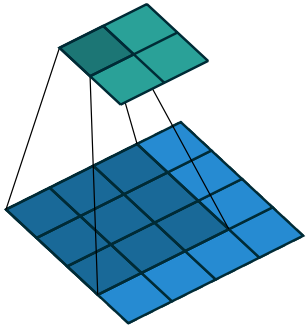


# 1-D convolution in deep networks

In pytorch, this is `torch.nn.Conv1d`.

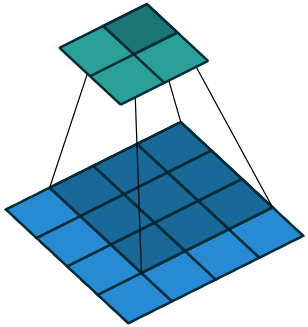
- As above, order reversed wrt "discrete convolution".
- Has many arguments; we'll explain them for 2-d convolution.
- Can also play with it via `torch.nn.functional.conv1d`.

# 2-D convolution in deep networks (pictures)



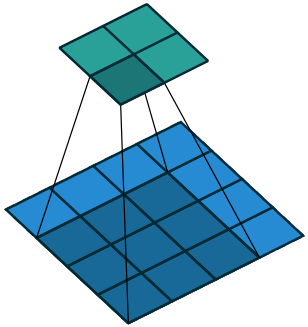
(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)



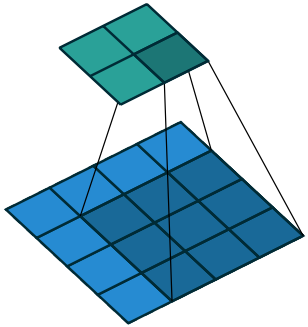
(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

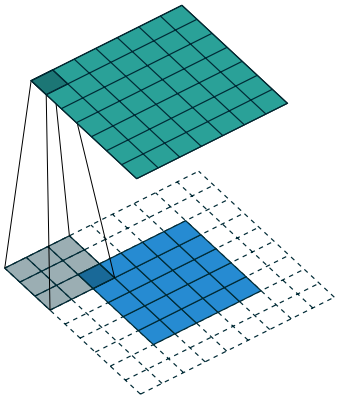
# 2-D convolution in deep networks (pictures)



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

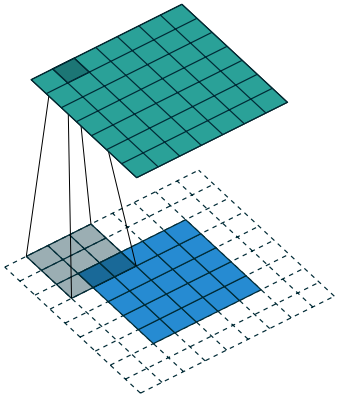
With padding.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

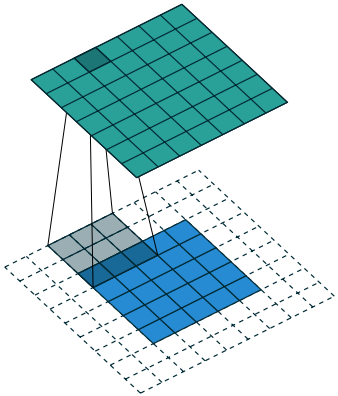
With padding.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

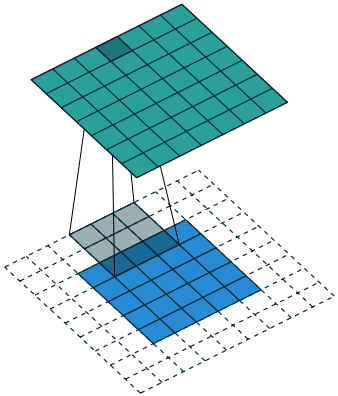
With padding.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

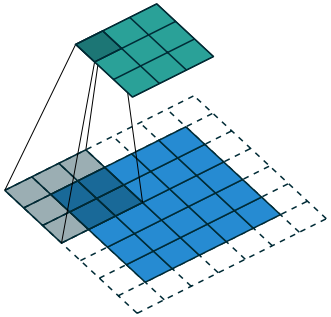
With padding.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

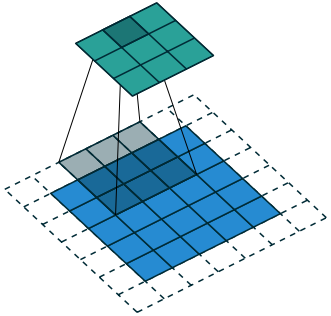
With padding, strides.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

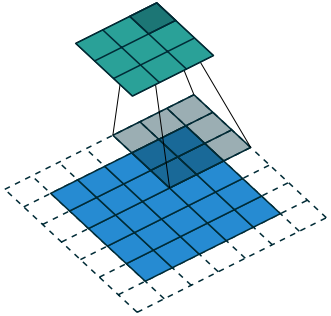
With padding, strides.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

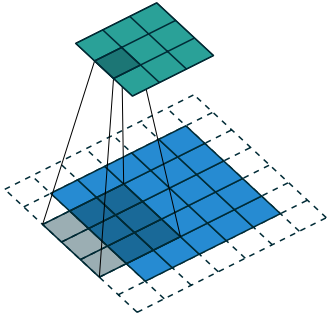
With padding, strides.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

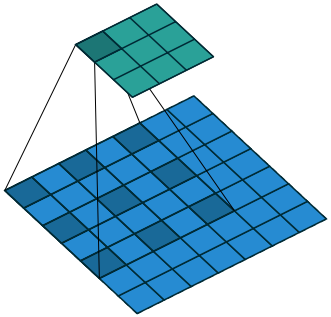
With padding, strides.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

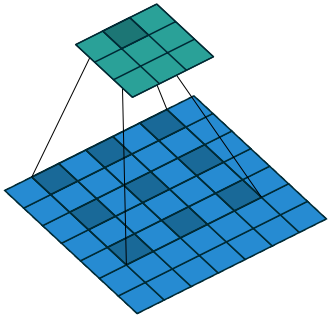
With dilation.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

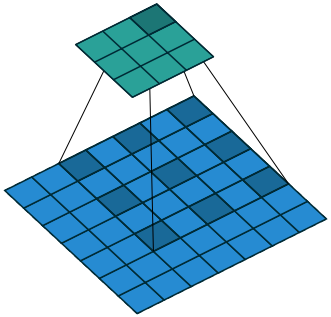
With dilation.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

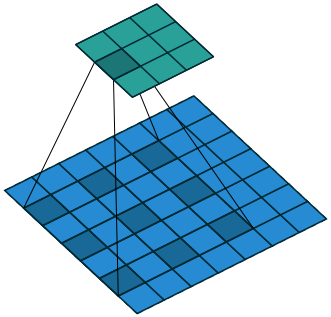
With dilation.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks (pictures)

With dilation.



(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# 2-D convolution in deep networks

- Invoke with `torch.nn.Conv2d`, `torch.nn.functional.conv2d`.
- Input and filter can have channels;
  a color image can have size $32 \times 32 \times 3$ for 3 color channels.
- Output can have channels;
  this means multiple filters.
- Other `torch` arguments: bias, stride, dilation, padding, ...
- Was motivated by computer vision community (primate V1);
  useful in Go, NLP, ...;
  many consecutive convolution layers leads to hierarchical structure.
- Convolution layers lead to major parameter savings over dense/linear layers.
- Convolution layers are linear!
  To check this, replace input $\boldsymbol{x}$ with $a\boldsymbol{x} + b\boldsymbol{y}$;
  the operation to make each entry of output is dot product, thus linear.
- Convolution, like ReLU, seems to appear in all major feedforward networks in past decade!

# Other standard layers

**Covered here:**

- Softmax.
- Max pooling.
- Batch normalization.
- Skip connections.

**Not covered here:**

- Other normalization layers.
- Blocks, not layers: e.g., attention blocks.
- ...

# Softmax

Replace vector input $\boldsymbol{z}$ with $\boldsymbol{z}' \propto e^{\boldsymbol{z}}$, meaning

$$\boldsymbol{z} \mapsto \left( \frac{e^{z_1}}{\sum_j e^{z_j}}, \ldots, \frac{e^{z_k}}{\sum_j e^{z_j}}, \right).$$

- Converts input into a probability vector.
- Typically the final layer of a network;
  useful for interpreting output network output as $\Pr[Y = y | X = x]$.
  Correspondingly, the inputs to this final softmax
  are sometimes called logits.
- If some coordinate $j$ of $\boldsymbol{z}$ dominates others, then softmax is close to $\mathbf{e}_j$.

# Cross-entropy loss with softmax baked in

Given two probability vectors $\boldsymbol{p}, \boldsymbol{q} \in \Delta_d := \left\{ \boldsymbol{v} \in \mathbb{R}^k : \boldsymbol{v} \geq 0, \sum_i v_i = 1 \right\}$, define **cross-entropy**

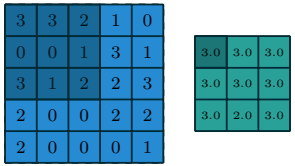
$$H(p, q) := -\sum_{i=1}^{k} p_i \ln q_i.$$

In `pytorch`, `cross_entropy(yhat, y)` takes logits $\hat{\boldsymbol{y}} \in \mathbb{R}^k$ and class label $y \in [k]$, and computes

$$\ell_{\mathsf{ce}}(\hat{\boldsymbol{y}}, y) = H(\boldsymbol{e}_y, \mathtt{softmax}(\hat{\boldsymbol{y}})) = \ln \left( \frac{\sum_i \exp(\hat{y}_i)}{\exp(\hat{y}_y)} \right)$$

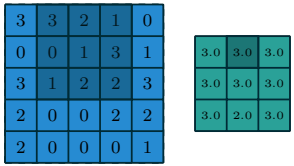
$$= -\hat{y}_y + \ln \left( \sum_i \exp(\hat{y}_i) \right).$$

# Max pooling



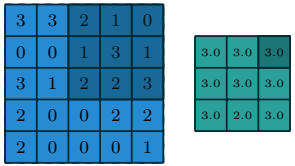

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# Max pooling



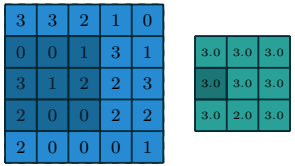

(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# Max pooling




(Taken from https://github.com/vdumoulin/conv_arithmetic by Vincent Dumoulin, Francesco Visin.)

# Max pooling




(Taken from `https://github.com/vdumoulin/conv_arithmetic` by Vincent Dumoulin, Francesco Visin.)

- Often used together with convolution layers; shrinks/downsamples the input.
- Another variant is average pooling.
- Implementation: `torch.nn.MaxPool2d` .

# Batch normalization

**Standardize** node outputs:

$$x \mapsto \left( \frac{x - \mu}{\sigma} \right) \cdot \gamma + \beta,$$

where the four parameters (!) have the following meaning.

- During training (by invoking `.train()`), $\mu$ is the mean over the minibatch (i.e., `x.mean(dim = 0, keepdim = True)`), $\sigma$ is the standard deviation over the minibatch. Additionally, running estimates of $\mu$ and $\sigma$ are updated.
- Outside training (by invoking `.eval()`), $\mu$ and $\sigma$ use the running estimates.
- $\gamma$ and $\beta$ are trainable affine parameters (i.e., they appear in `.parameters()`). Last I checked, the defaults for having them on or off differed between pytorch and tensorflow.

Further notes.

- Relevant classes are `torch.nn.BatchNorm{1d,2d,3d}` based on input shape.
- Transformers use `torch.nn.LayerNorm`, which is similar.
- BatchNorm is not well understood, but seems to help both optimization and generalization.

# Dropout

Given a parameter $p \in [0, 1]$ (corresponding `torch.nn.Dropout(p)`):

1. During training (by invoking `.train()`), for each input coordinate $x$, sample $b \sim \text{Bernoulli}(1 - p)$, and output $\frac{xb}{1-p}$.
2. During testing (by invoking `.eval()`), perform the identity mapping.

Remarks.

1. It is believed to regularize.
2. It is sometimes believed to promote redundancy in the network and a certain favorable "averaging" or "aggregation" effect (see later lectures).
3. It stopped being popular outside text domains for many years; transformers use it by default (I'm not sure about transformers for vision), so here it is.

# Skip connections

- Can model resnet as a sequence of blocks computing

$$\boldsymbol{z} \mapsto \boldsymbol{z} + f_i(\boldsymbol{z}),$$

  where a typical $f_i$ is convolution, batchnorm, relu convolution, relu.
- One standard practice is to have a linear layer at the end of $f_i$, and initialize it to zero (normally bad); this way, this subnetwork is identity by default.
- These $f_i$'s are "residuals".
- The identity connections are sometimes called "skip connections".
- There are many variants of the idea (e.g., DenseNet).
  Modern networks have connections all over the place.

# Gradient descent and “back-propagation”

# Gradient descent and "back-propagation"

- Deep networks are generally trained with gradient-based methods.
- We will only give an abstract presentation, since `pytorch` lets you not worry about it:
  - As in the pytorch tutorial, you mark certain tensors with `requires_grad = True`, which means "I care about gradients with respect to this object". Note that `requires_grad = True` is a bit fragile, and defaults to `False` on temporaries created within `torch.no_grad()` contexts.
  - Whenever expressions appear with that object, pytorch keeps track of this (in a big "computation graph"), unless it is within a `torch.no_grad()` context.
  - If at some later point you invoke `expression.backward()`, pytorch unrolls all the computation used to construct `expression`, populating the `.grad` member for every tensor which has `requries_grad = True`.
- There is little understanding of why gradient descent works for deep networks.

# Warm-up: gradients for linear prediction

Suppose

$$\widehat{\mathcal{R}}(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^{n} \ell(y_i \boldsymbol{x}_i^{\mathsf{T}} \boldsymbol{w}).$$

- Gradient passes through average; focus on one example:

$$\nabla_{\boldsymbol{w}} \ell(y \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}).$$

- By the chain rule,

$$\nabla_{\boldsymbol{w}} \ell(y \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}) = \ell'(y \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}) y \boldsymbol{x}.$$

- Abstraction: define $p(\boldsymbol{w}) := y \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}$, and note

$$\nabla \ell(y \boldsymbol{x}^{\mathsf{T}} \boldsymbol{w}) = \nabla \ell(p(\boldsymbol{w})) = \frac{\partial \ell(p(\boldsymbol{w}))}{\partial p(\boldsymbol{w})} \frac{\partial p(\boldsymbol{w})}{\partial \boldsymbol{w}}.$$

Let's iterate this to handle deep networks. (We will be lazy and abstract and not worry about shapes.)

Fix example $(\boldsymbol{x}, y)$, and define

$$\begin{aligned}\boldsymbol{p}_1 &:= \sigma_1(\boldsymbol{W}_1\boldsymbol{x}),\\ \boldsymbol{p}_{i+1} &:= \sigma_{i+1}(\boldsymbol{W}_{i+1}\boldsymbol{p}_i),\\ \boldsymbol{p}_L &:= \ell(y\sigma_L(\boldsymbol{W}_L\boldsymbol{p}_{L-1})).\end{aligned}$$

Then

$$\nabla_{\boldsymbol{W}_i}\ell(yf(\boldsymbol{x};\boldsymbol{w})) = \frac{\partial \boldsymbol{p}_L}{\partial \boldsymbol{p}_i}\frac{\partial \boldsymbol{p}_i}{\partial \boldsymbol{W}_i}, \qquad \text{where } \frac{\partial \boldsymbol{p}_L}{\partial \boldsymbol{p}_i} = \frac{\partial \boldsymbol{p}_L}{\partial \boldsymbol{p}_{i+1}}\frac{\partial \boldsymbol{p}_{i+1}}{\partial \boldsymbol{p}_i}.$$

- **Forward pass:** compute $\boldsymbol{p}_L$ by computing $\boldsymbol{p}_1$ and moving up inductively.
- **Backward pass:** compute gradients from $\boldsymbol{W}_L$ down to $\boldsymbol{W}_1$ via the inductive chain rule.
- Some people say "backprop is just chain rule", but in fact it is the computational suggestion to use this two-stage forward/backward recursion.
- Same procedure generalizes to any acyclic computation graph (skip connections, etc.).
- In pytorch:
  - When you construct expressions in pytorch, it immediately computes their output (i.e., immediately and incrementally builds the forward pass), and stores pointers within tensors to unroll the computation and perform backward, if necessary.
  - If you invoke `.backward()` later, it follows these backward pointers, and every tensor with `requires_grad = True` has its `.grad` member populated with the partial derivative as above.

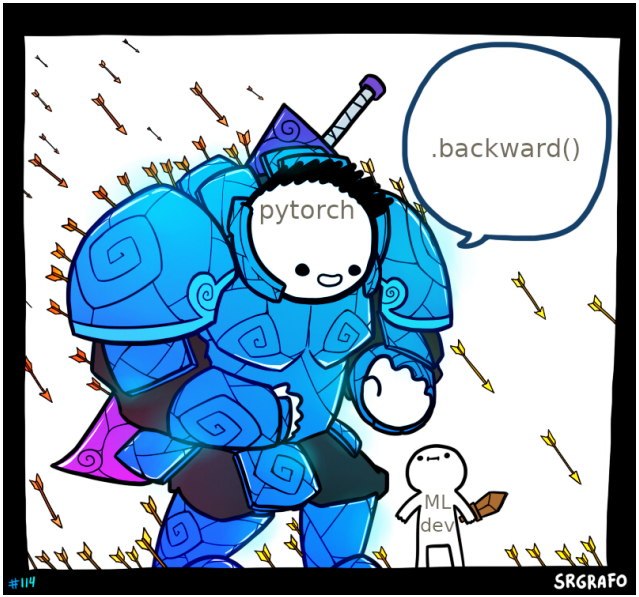
# Vanishing/exploding gradients

Note the term

$$\nabla_{\boldsymbol{W}_i} \ell(yf(\boldsymbol{x}; \boldsymbol{w})) = \frac{\partial \boldsymbol{p}_L}{\partial \boldsymbol{p}_i} \frac{\partial \boldsymbol{p}_i}{\partial \boldsymbol{W}_i}, = \prod_{j=i}^{L-1} \frac{\partial \boldsymbol{p}_{j+1}}{\partial \boldsymbol{p}_j}.$$

With many-layered networks, this computation easily **explodes** (becomes large) or **vanishes** (goes to zero).

Many architecture and optimization choices are designed to mitigate this.




"Did you compute your gradients correctly?" — Leon Bottou.

# Initialization

Standard deep networks do not initialize $\boldsymbol{W}_i = 0$.

- Need to break symmetry: initialize in some way so that all gradients different.
- Random initialization is common; usually it is formalized as $\mathcal{N}(0, 1/d_{i-1})$ per coordinate.
- `torch.nn.Linear()` documentation includes defaults (as of 2022 and the few previous years, it is uniform in $[-1/\sqrt{d_{i-1}}, +1/\sqrt{d_{i+1}}]$, which doesn't quite match variance to the Gaussian case...
- Many other tricks;

  e.g., batch norm initialization used to force identity residual blocks.

  Uniform and "truncated Gaussian" are common.

# Learning rates

- Lots of magic here. Often $\eta \in \{0.001, 0.01, 0.1\}$ suffices.
- Modern networks use complicated **learning rate schedules**; e.g., following exponential decay, or a triangle, or oscillating.
  See `torch.optim.lr_scheduler` for standard choices.

# Other optimization details

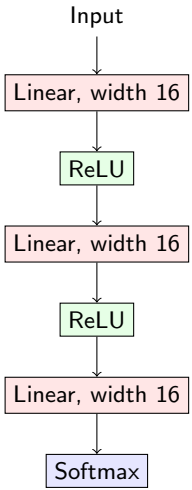
- Minibatch size seems magic and hardware/problem dependent.
- Terminology: "epoch" is a full pass over training data, but an iteration or step is the update with a single minibatch.
- See `torch.optim` for many options; Adam is a modern standard.

# Data augmentation

**Data augmentation** is a standard part of deep network training:

- Rather than training on just the provided training set, use a bunch of computation to generate new reasonable images.
- For example, for vision tasks, it's common to include random crops, flips.
- It can give a shockingly large bump in performance, and therefore is standard, and quite complicated in modern setups.

# Deep network diagrams revisited

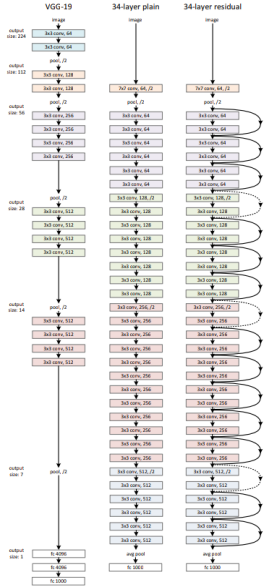



```
torch.nn.Sequential(
    torch.nn.Linear(2, 3, bias =
        True),
    torch.nn.ReLU(),
    torch.nn.Linear(3, 4, bias =
        True),
    torch.nn.ReLU(),
    torch.nn.Linear(4, 2, bias =
        True),
)
```

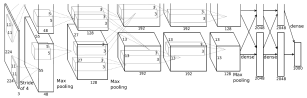
Remarks.

- Diagram format is not standard.
- As long as anyone can unambiguously reconstruct the network, it's fine.
- Remember that edges can transmit full tensors now!

# Other example diagrams




Taken from ResNet paper, 2015.



AlexNet paper, 2012.

# pytorch meta-algorithm.

1. Clean/augment data.

   **Data augmentation** crucial in deep networks; see lecture 10.

2. Pick model/architecture.

   ResNet, DenseNet, WideResnet...

3. Pick a loss function measuring model fit to data.

   Typically cross-entropy for classification, but many sophisticated choices in modern networks.

4. Run a gradient descent variant to fit model to data.

   Adam and SGD most common, both with minibatches.

5. Tweak 1-4 until training error is small.

6. Tweak 1-5, possibly reducing model complexity, until testing error is small.

   "Model complexity" is not well understood with deep networks; e.g., increasing width can reduce test error. A typical way is to increase regularization, data augmentation, and pick a simpler model.

# Summary for today

1. Basic deep network definition.
2. Convolutions and other modern standards.
3. Gradient descent and "back-propagation".
4. `pytorch` code and meta-algorithm.

(Appendix.)

# Miscellanea

- Adversarial examples: on some vision tasks, these networks seem on par with human perception (in terms of training and test error).

  However, there training points which can be imperceptibly perturbed so that the class label flips! In this way, they are nothing like human perception.

  Since deep networks are rolling out in many human-facing applications, these examples are scary, and constitute a major area of research.

# Miscellanea

- Adversarial examples: on some vision tasks, these networks seem on par with human perception (in terms of training and test error).

  However, there training points which can be imperceptibly perturbed so that the class label flips! In this way, they are nothing like human perception.

  Since deep networks are rolling out in many human-facing applications, these examples are scary, and constitute a major area of research.

- Feature extraction: we can train a network on some huge data, chop it in the middle, and use these features as input to train a network on some other task, in particular one with much less data.

  (The deep learning community sometimes calls this transfer learning; which more generally means transfering information from one prediction task to another.)

# Miscellanea

- Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l)$, e.g., an english sentence? We can have a network which eats this sequence one by one; for $\boldsymbol{x}_i$, it also consumes a previous state $\boldsymbol{s}_i$, and outputs $\boldsymbol{s}_{i+1}$.

  Many natural language processing (NLP) tasks now use RNNs.

# Miscellanea

- Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l)$, e.g., an english sentence? We can have a network which eats this sequence one by one; for $\boldsymbol{x}_i$, it also consumes a previous state $\boldsymbol{s}_i$, and outputs $\boldsymbol{s}_{i+1}$.

  Many natural language processing (NLP) tasks now use RNNs.

- Dynamic networks and differentiable programming. In the early code subclassing `torch.nn.Module`, we could have made the `forward` function do something more complicated; e.g., the number of layers can be variable.

  In `pytorch`, differentiable programming can concretely mean `forward` functions that look closer to full Turing Machines.

# Miscellanea

- Recurrent networks (RNNs). What should we do if our input is some arbitrary length sequence $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l)$, e.g., an english sentence? We can have a network which eats this sequence one by one; for $\boldsymbol{x}_i$, it also consumes a previous state $\boldsymbol{s}_i$, and outputs $\boldsymbol{s}_{i+1}$.

  Many natural language processing (NLP) tasks now use RNNs.
- Dynamic networks and differentiable programming. In the early code subclassing `torch.nn.Module`, we could have made the `forward` function do something more complicated; e.g., the number of layers can be variable.

  In `pytorch`, differentiable programming can concretely mean `forward` functions that look closer to full Turing Machines.
- Architecture search. Since the original work on neural networks, there have been attempts to automatically search for architectures. The bottom line is that it seems such methods waste computation when compared with simple trying 5-10 architectures and training them longer; but maybe it will change.

# Miscellanea

- GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly).

  In `pytorch`, you can send `torch.nn.Module` instances to GPU with `.cuda()` or `.to()`, just as with tensors.

  GPUs are fast when you feed them big tensor operations. (E.g., write `((X @ w - y).norm() *** 2).mean()`, not a loop.) Moving things between CPU and GPU is slow.

# Miscellanea

- GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly).

  In `pytorch`, you can send `torch.nn.Module` instances to GPU with `.cuda()` or `.to()`, just as with tensors.

  GPUs are fast when you feed them big tensor operations. (E.g., write `((X @ w - y).norm() *** 2).mean()`, not a loop.) Moving things between CPU and GPU is slow.
- Dropout is a regularization technique that involves randomly zeroing the outputs of nodes during training. It is less popular than it used to be, but still in use for certain applications (e.g., NLP).

# Miscellanea

- GPUs can process thousands of simple floating point operations in parallel, and massively speed up many of the computations here (my GPU machine is 100x faster than my laptop when I set things up correctly).

  In `pytorch`, you can send `torch.nn.Module` instances to GPU with `.cuda()` or `.to()`, just as with tensors.

  GPUs are fast when you feed them big tensor operations. (E.g., write `((X @ w - y).norm() *** 2).mean()`, not a loop.) Moving things between CPU and GPU is slow.
- Dropout is a regularization technique that involves randomly zeroing the outputs of nodes during training. It is less popular than it used to be, but still in use for certain applications (e.g., NLP).
- It is typically stated that deep networks are data hungry. I'm not sure if that's a necessity, or merely a consequence of our current training practices.

# Miscellanea

- History. Deep networks data back to the 1940s; the original "training algorithms" consisted of a human manually setting weights.

  They have come and gone multiple times. This phase is the first time they were reliably trainable with so many layers. I'm not sure why, but the reasons include: access to more data, GPUs (ResNet training is very slow), ReLU, random initialization, "social programming" and a generally healthy software ecosystem, ...

# Supplemental reading

The new edition of the Murphy book (available online) has three chapters covering some of these deep network topics, and a few not covered here.