Computer Science

CSC580: Principles of Data Science

Feed Forward Neural Networks

Jason Pacheco

Basis Functions

Basis functions transform linear models into nonlinear ones…

Linear Regression

$$y = w^T x$$

$$\downarrow$$

$$y = w^T \phi(x)$$

Classification
(Logistic Regression)

$$y = \sigma(w^T x)$$

$$\downarrow$$

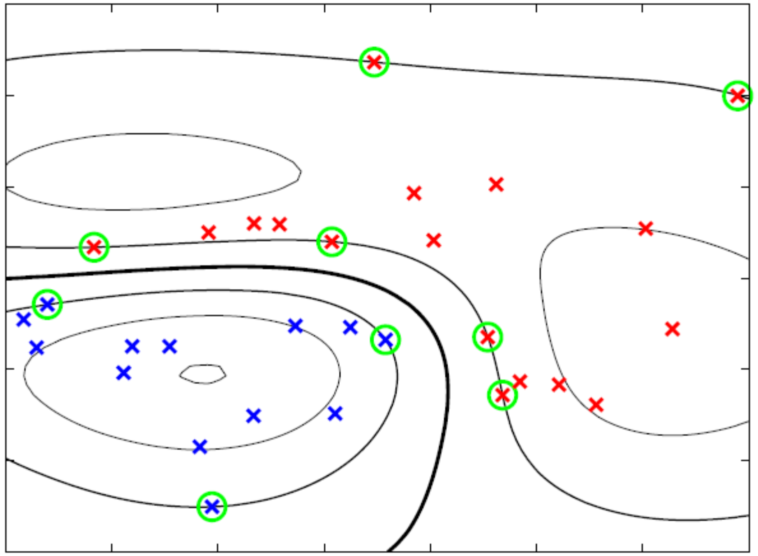
$$y = \sigma(w^T \phi(x))$$

…but it is often difficult to find a good basis transformation

Learning Basis Functions

What if we could learn a basis function so that a simple linear model performs well…

Data Space

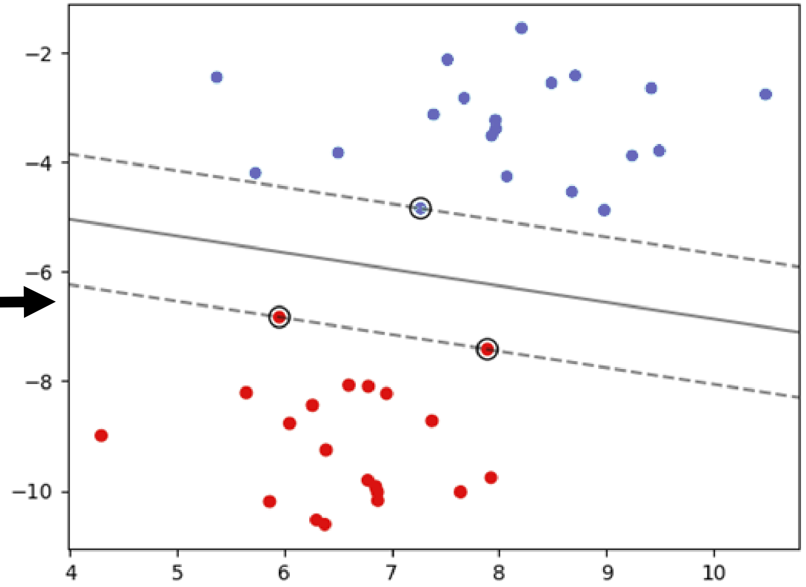


Ignore the circled points…I reused these from the SVM slides

Neural Net

$\phi(x)$

Warped Space



…this is essentially what standard neural networks do…

Neural Networks

- Flexible nonlinear transformations of data
- Resulting transformation is easily fit with a linear model
- Relatively efficient learning procedure scales to massive data
- Apply to many Machine Learning / Data Science problems
  - Regression
  - Classification
  - Dimensionality reduction
  - Function approximation
  - Many application-specific problems

Neural Networks

Forms of NNs are used all over the place nowadays…

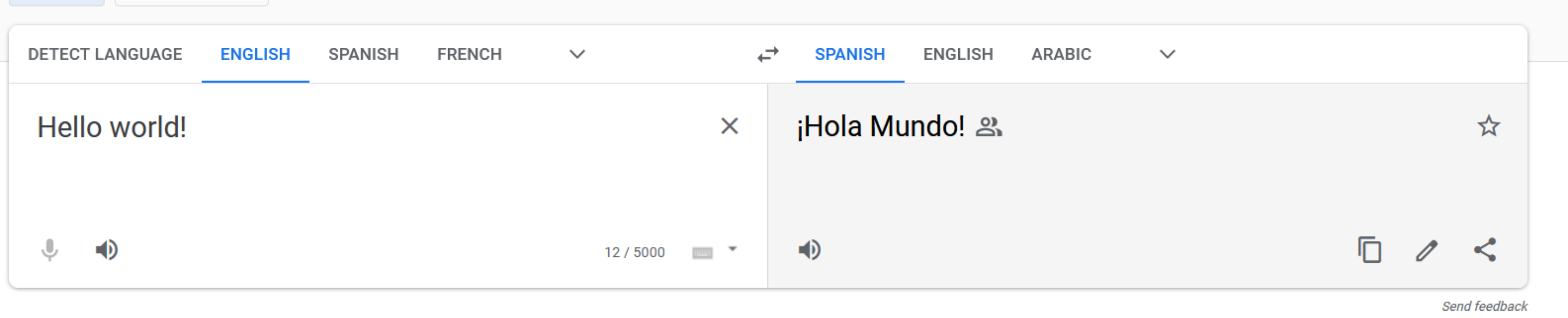
ChatGPT

AI Chat Bots

Self-Driving Cars

Creepy Robots

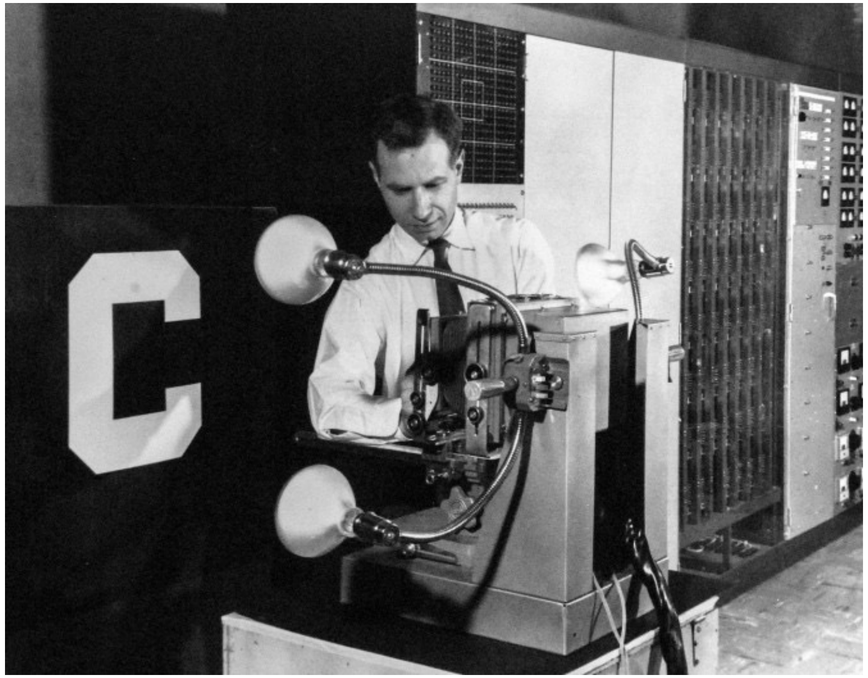
Machine Translation



Rosenblatt's Perceptron

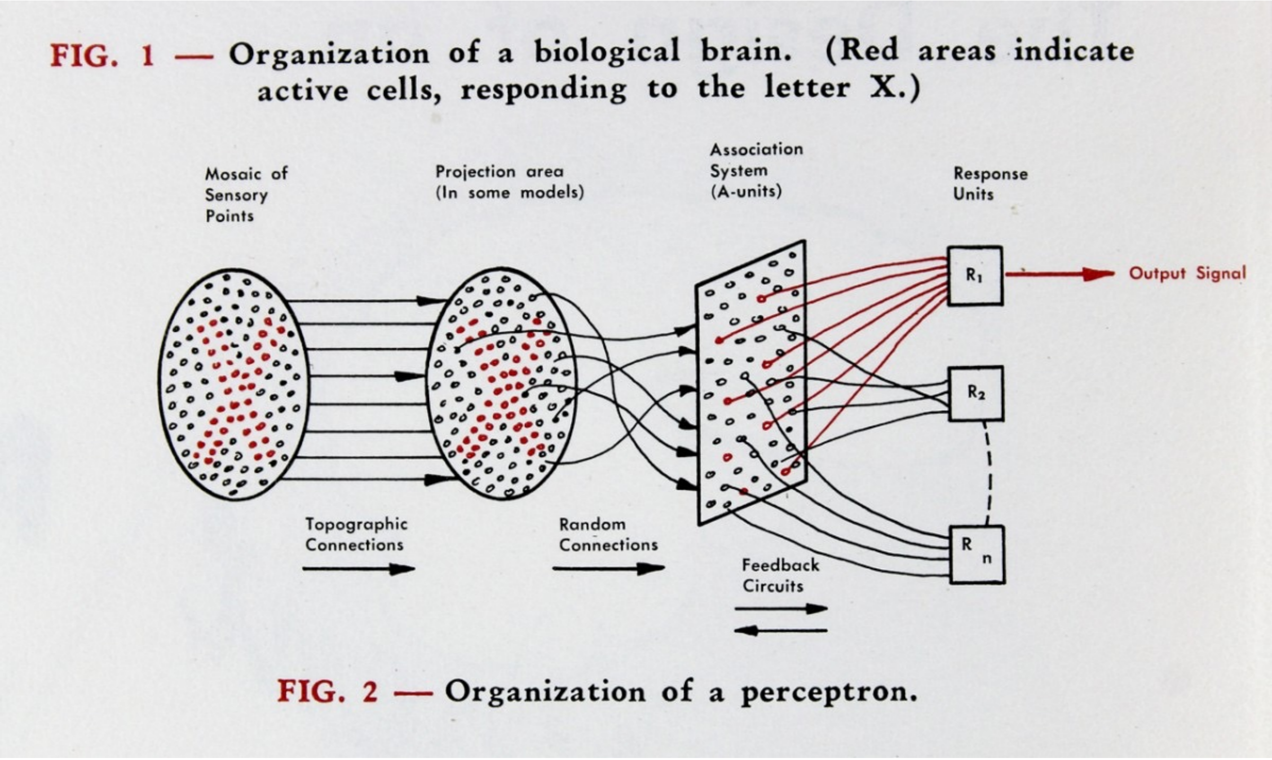
Despite recent attention, neural networks are fairly old

In 1957 Frank Rosenblatt constructed the first (single layer) neural network known as a "perceptron"

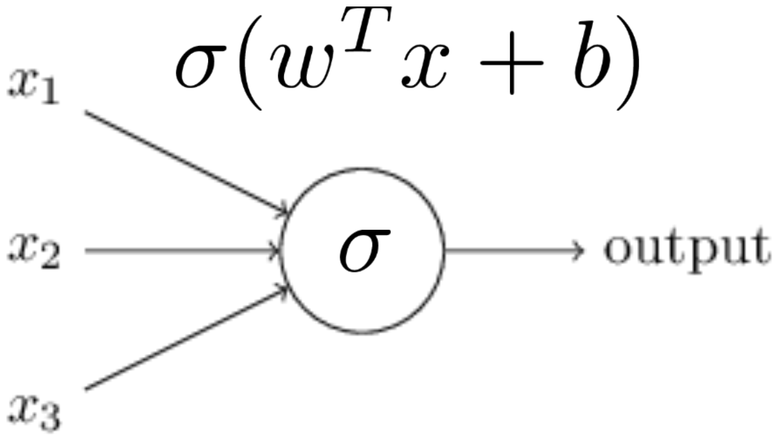


He demonstrated that it is capable of recognizing characters projected onto a 20x20 "pixel" array of photosensors

Rosenblatt's Perceptron

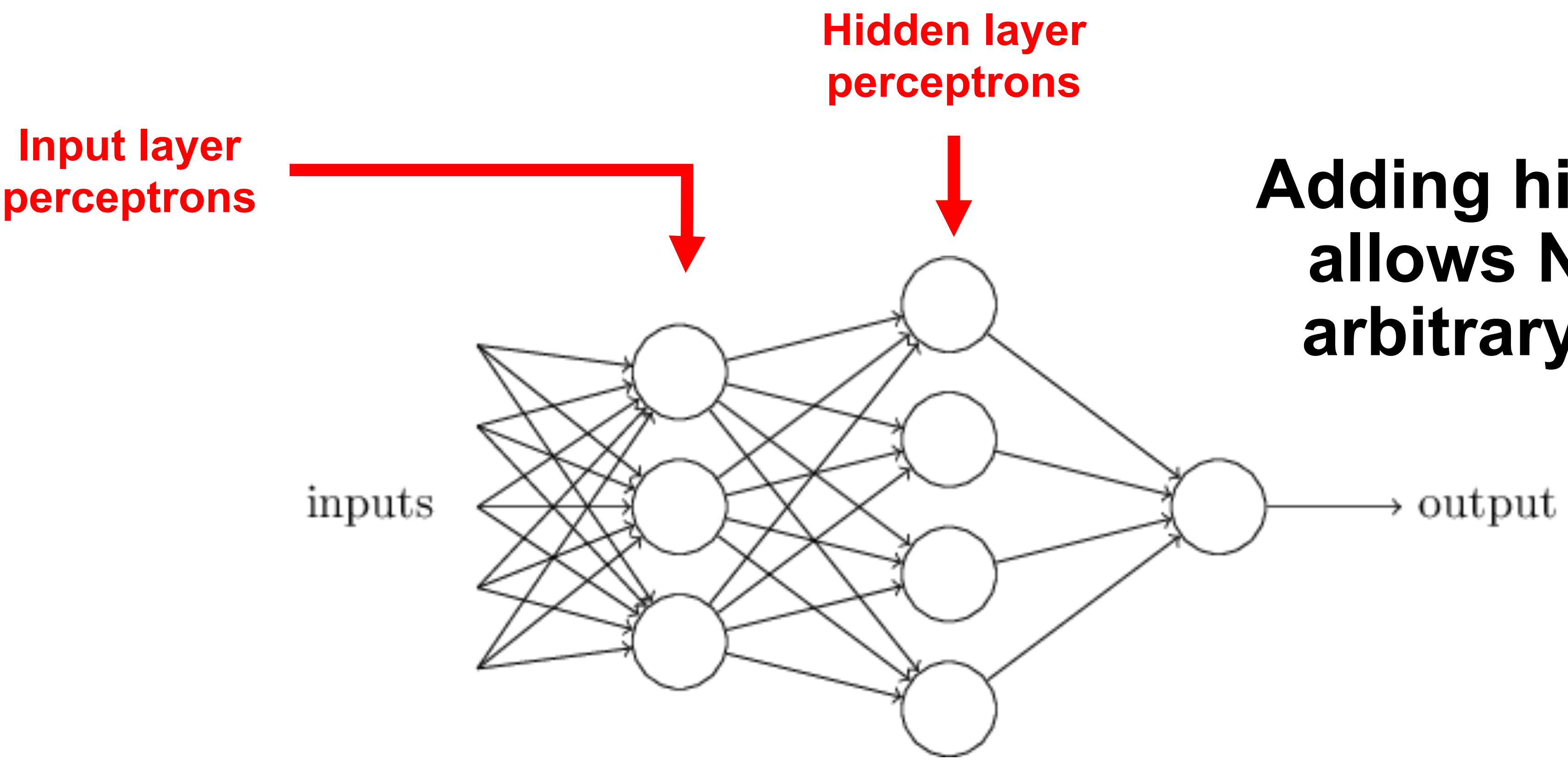


Perceptron



- In Rosenblatt's perceptron, the inputs are tied directly to output
- "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanics" (1962)
- Criticized by Marvin Minsky in book "Perceptrons" since can only learn linearly-separable functions
- **The perceptron is just logistic regression in disguise**

Multilayer Perceptron



Adding hidden layers allows NN to learn arbitrary functions

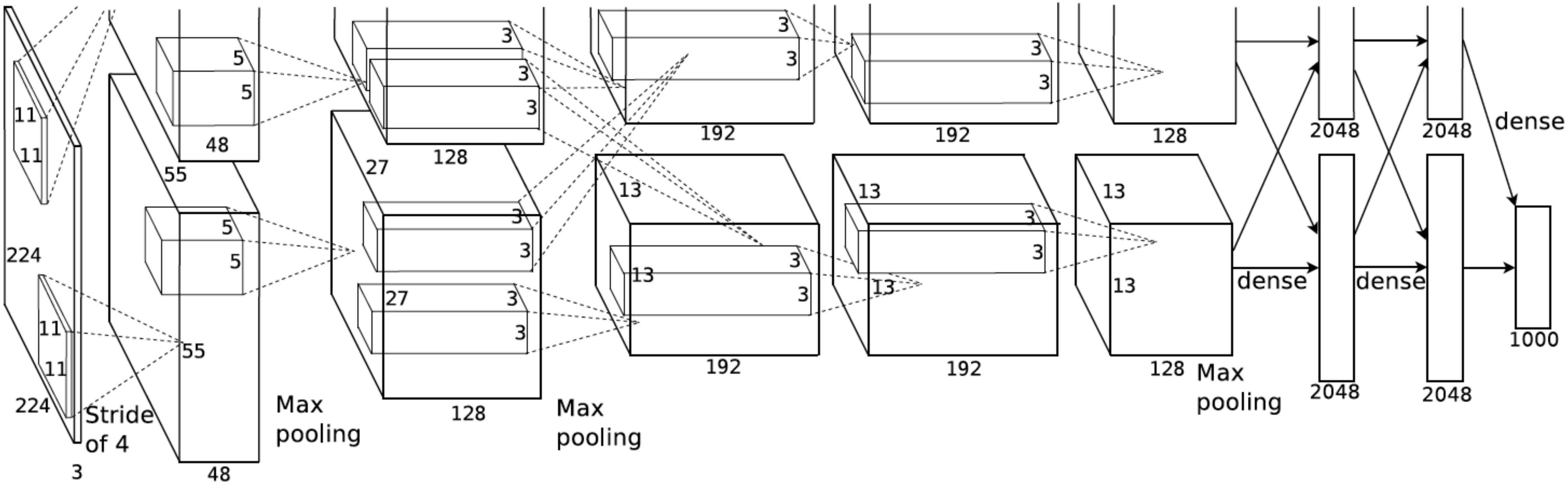
This is the quintessential *Neural Network*…

…also called *Feed Forward Neural Net* or *Artificial Neural Net*

[Source: http://neuralnetworksanddeeplearning.com]

Modern Neural Networks

Modern *Deep Neural networks* add many hidden layers



…and have many trillions of parameters to learn

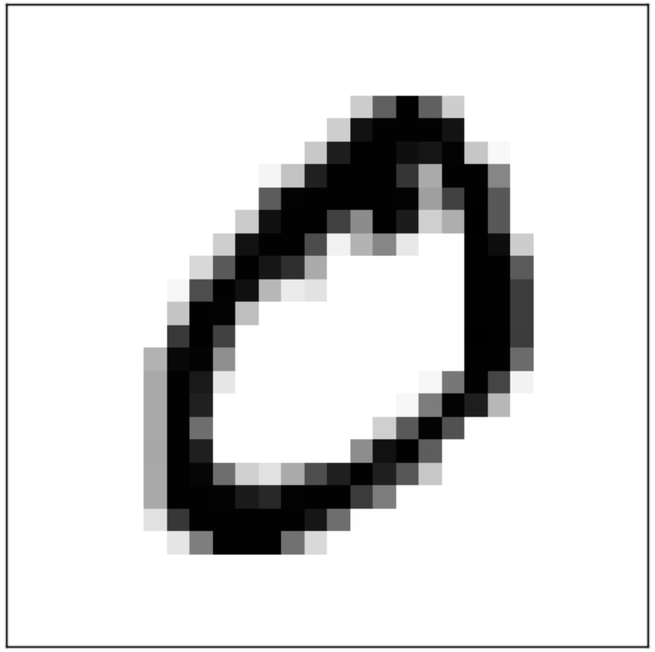
[Source: Krizhevsky et al. (NIPS 2012)]

Handwritten Digit Classification

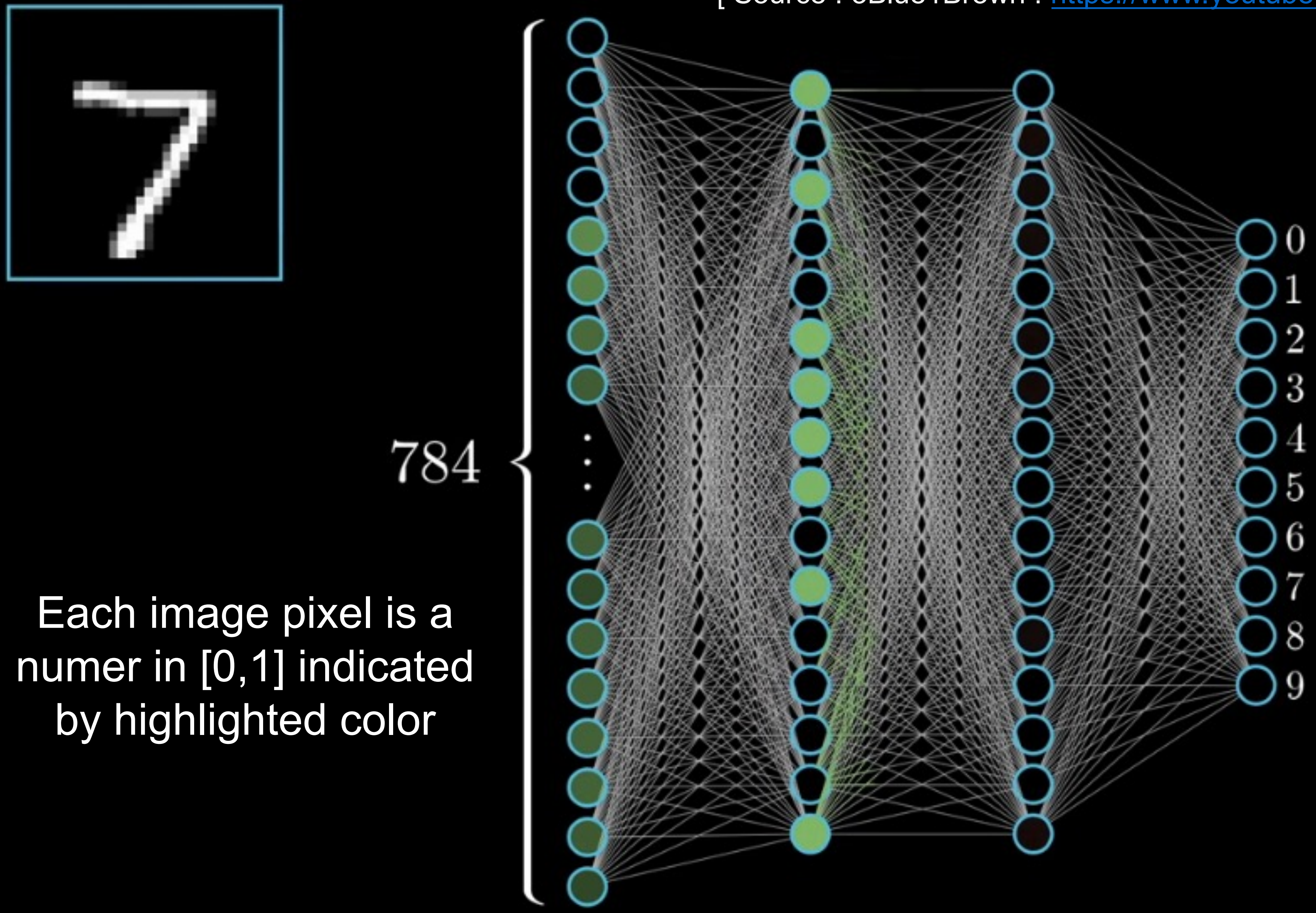
Classifying handwritten digits is the “Hello World” of NNs

Modified National Institute of Standards and Technology (MNIST) database contains 60k training and 10k test images

Each character is centered in a 28x28=784 pixel grayscale image

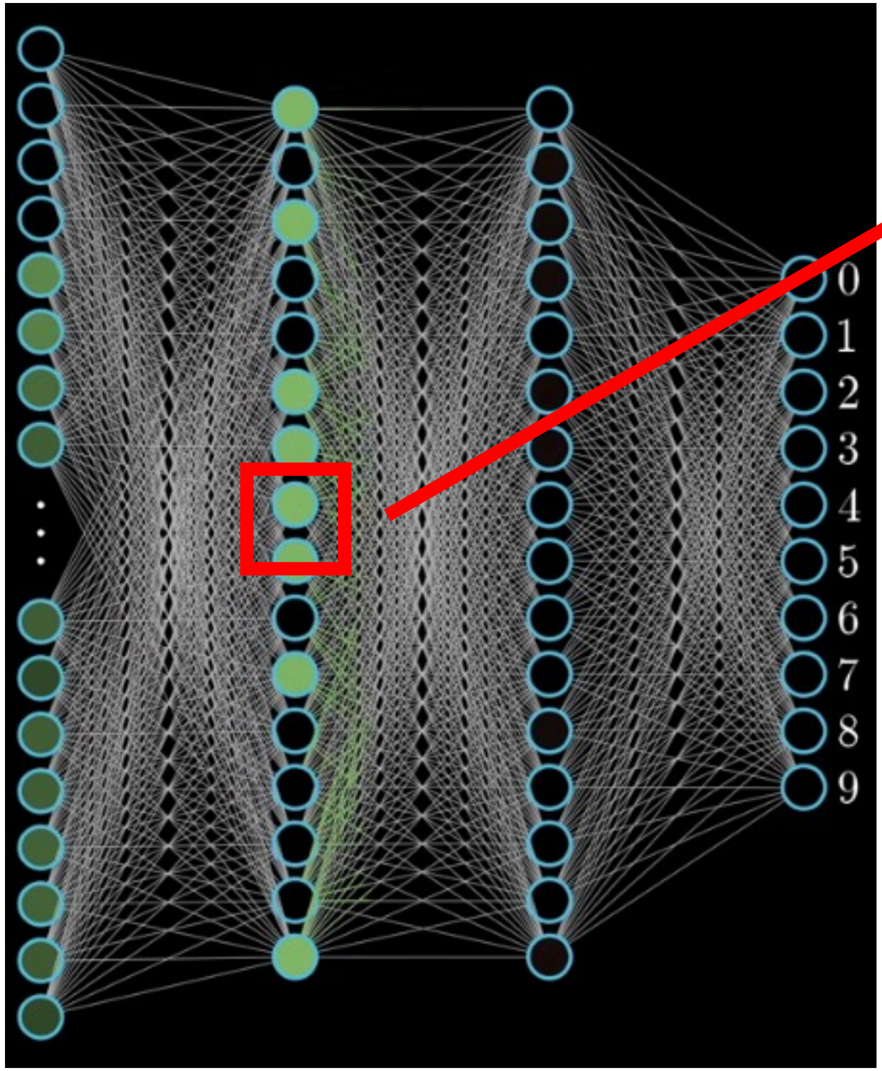


[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



Each image pixel is a numer in [0,1] indicated by highlighted color

Feedforward Procedure



Each node computes a *weighted combination* of nodes at the previous layer…

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

Then applies a *nonlinear function* to the result

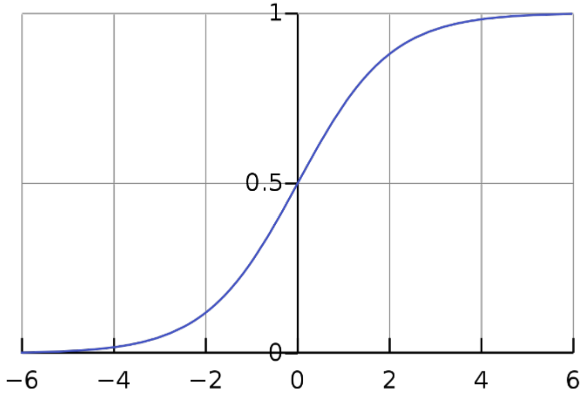
$$\sigma(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b)$$

Often, we also introduce a constant *bias* parameter

Nonlinear Activation functions

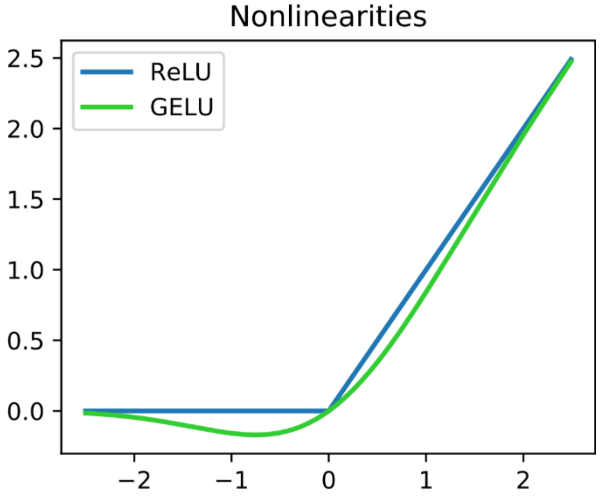
We call this an *activation function* and typically write it in vector form,

$$\sigma(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) = \sigma(w^T x + b)$$



An early choice was the *logistic function*,

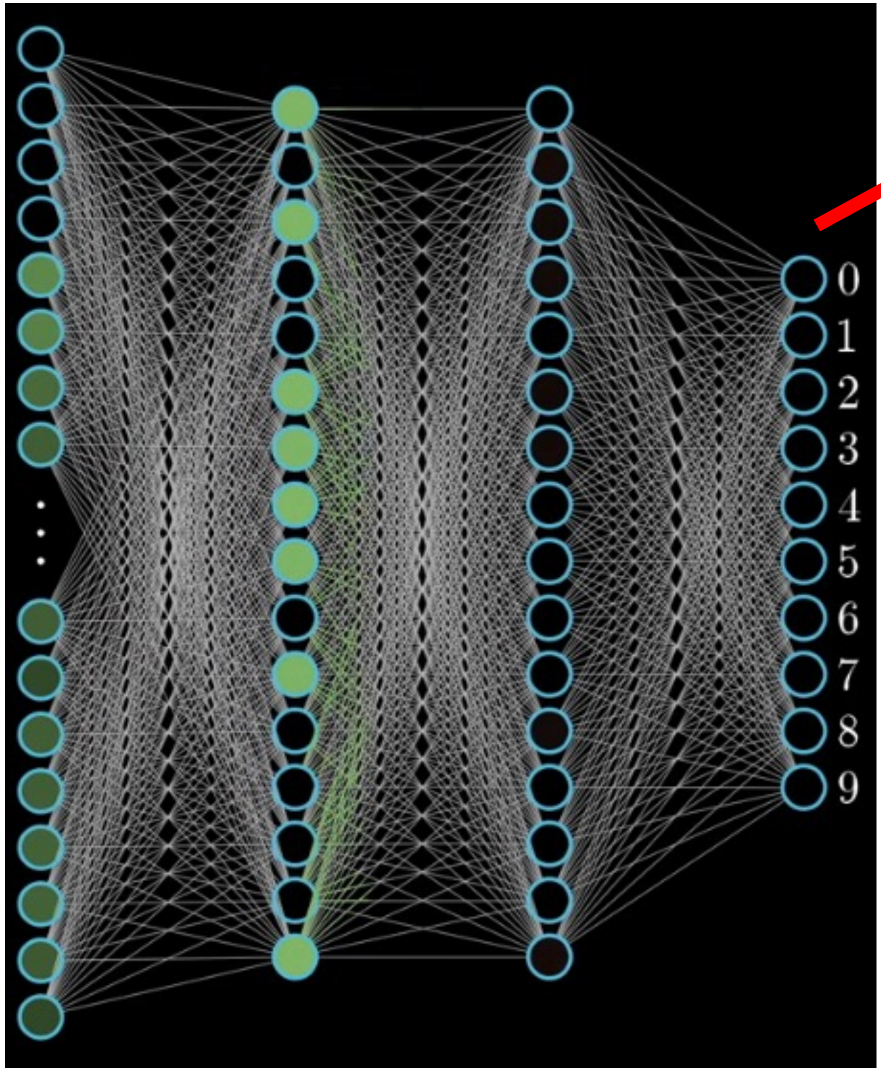
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$



Later found to lead to slow learning and *ridge functions* like the *rectified linear unit (ReLU),*

$$\sigma(w^T x + b) = \max(0, w^T x + b)$$

Multilayer Perceptron



Final layer is typically a linear model…for classification this is a Logistic Regression

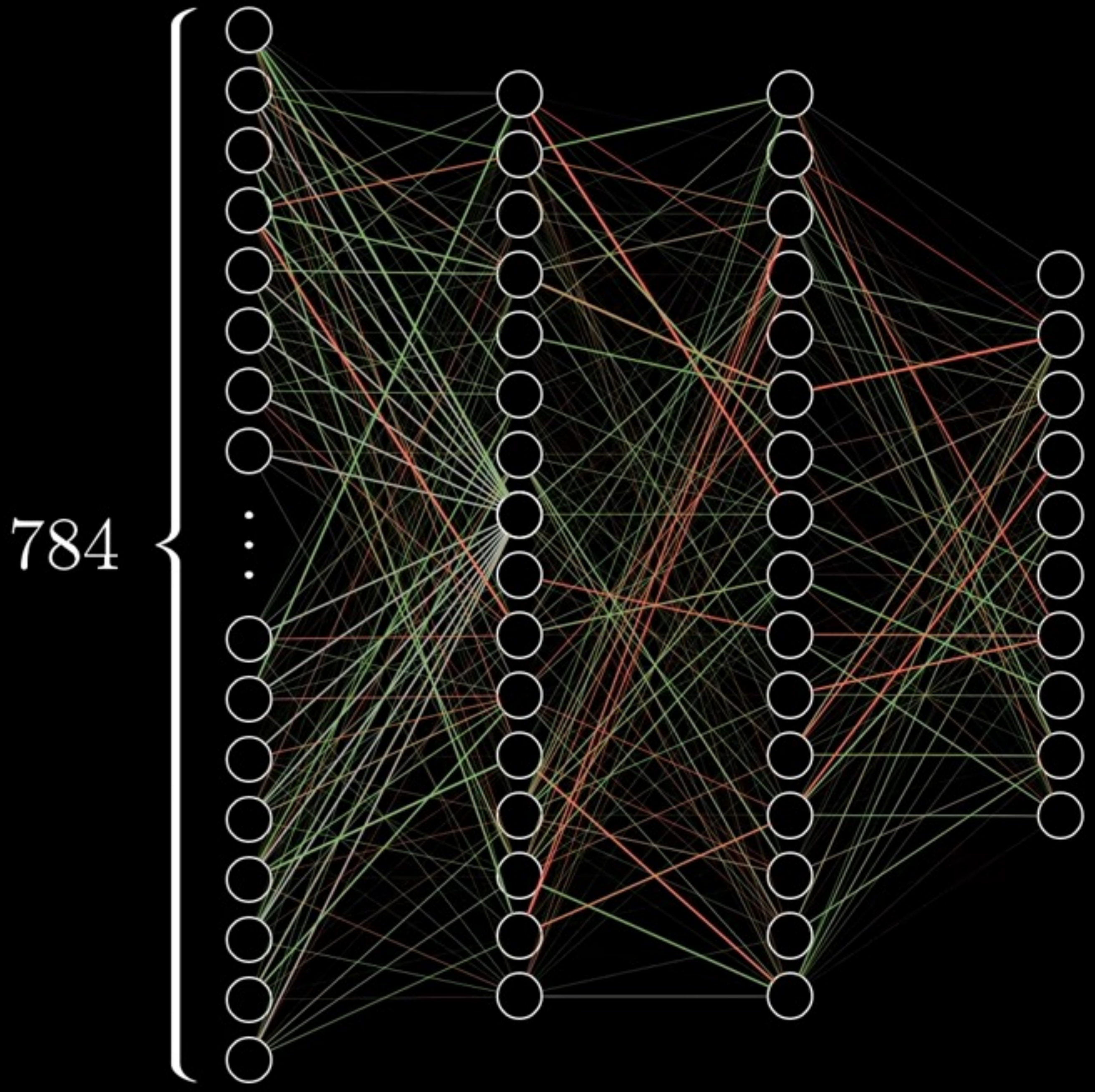
$$\sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Vector of activations from previous layer

Recall that for multiclass logistic regression with K classes,

$$p(\text{Class} = k \mid x) \propto \sigma(w_k^T x + b_k)$$

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



$$784 \times 16 + 16 \times 16 + 16 \times 10$$

weights

$$16 + 16 + 10$$

biases

13,002

Each parameter has some impact on the output…need to tweak (learn) all parameters simultaneously to improve prediction accuracy

Training Multilayer Perceptron

Our cost function for ith input is error in terms of weights / biases…

$$\mathrm{Cost}_i\underbrace{(w_1, \ldots, w_n, b_1, \ldots, b_n)}_{\text{13,002 Parameters in this network}}$$

…minimize cost over all training data…

$$\min_{w,b} \mathcal{L}(w, b) = \sum_i \mathrm{Cost}_i(w_1, \ldots, w_n, b_1, \ldots, b_n)$$

This is a super high-dimensional optimization (13,002 dimensions in this example)…how do we solve it?

Gradient descent!

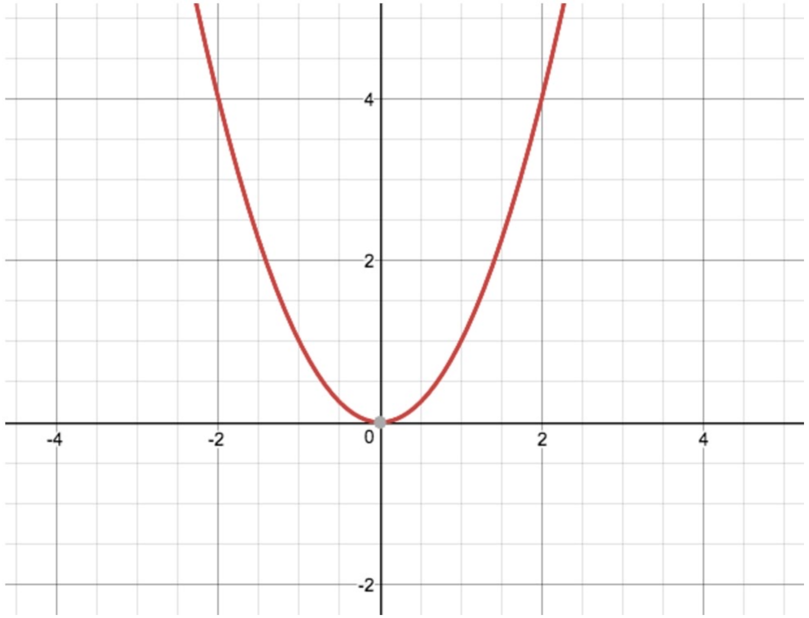
Learning algorithm intuition

- Gradient descent: Move in direction of greatest improvement
- “Knob turning”
  - ”knob” = weight of an edge
  - If a neuron increases the probability of an incorrect prediction, its knobs will be turned down.
  - If a neuron increases the probability of a correct prediction, its knobs will be turned up.

Training Multilayer Perceptron

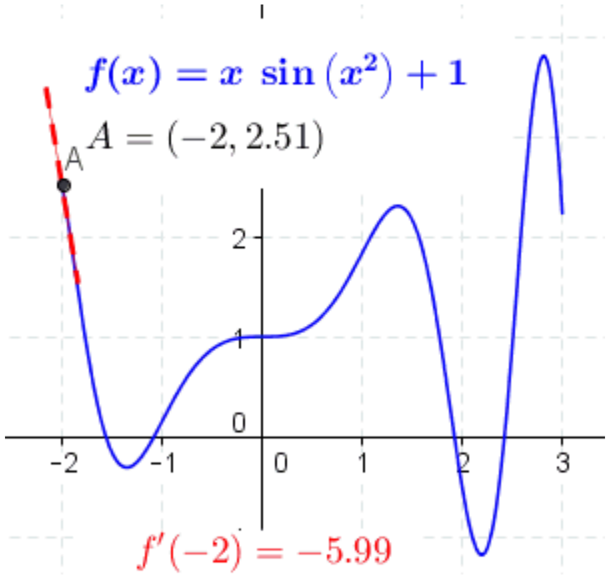
Need to find zero derivative (gradient) solution…

Convex Cost Function



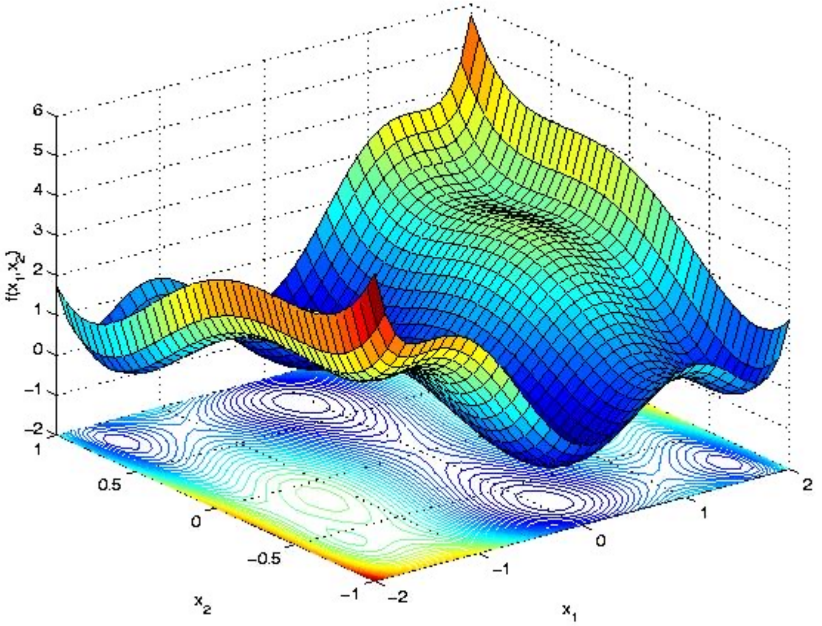
YAY!

Non-convex Cost Function



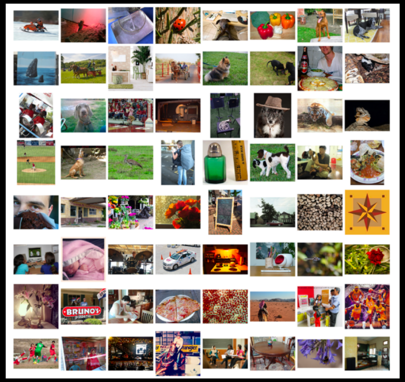
Boo!

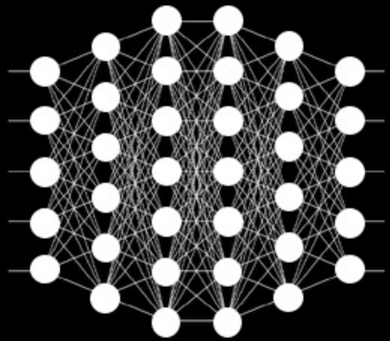
High-Dimensional Non-convex

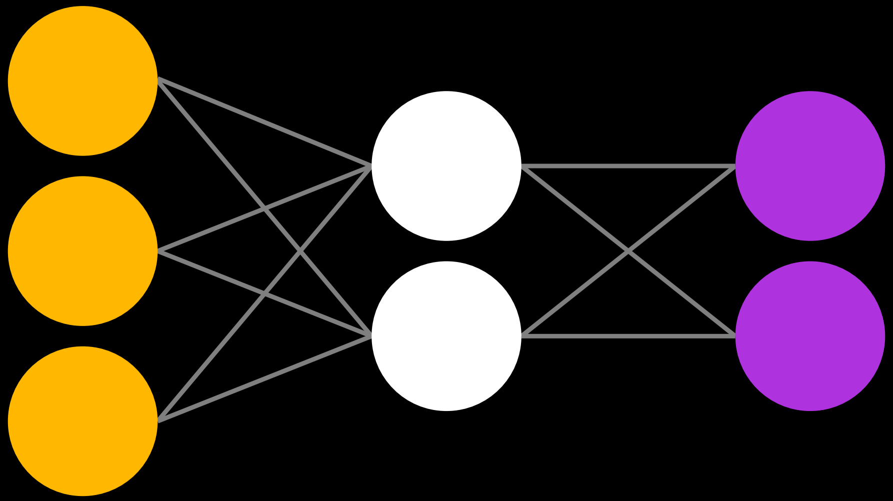


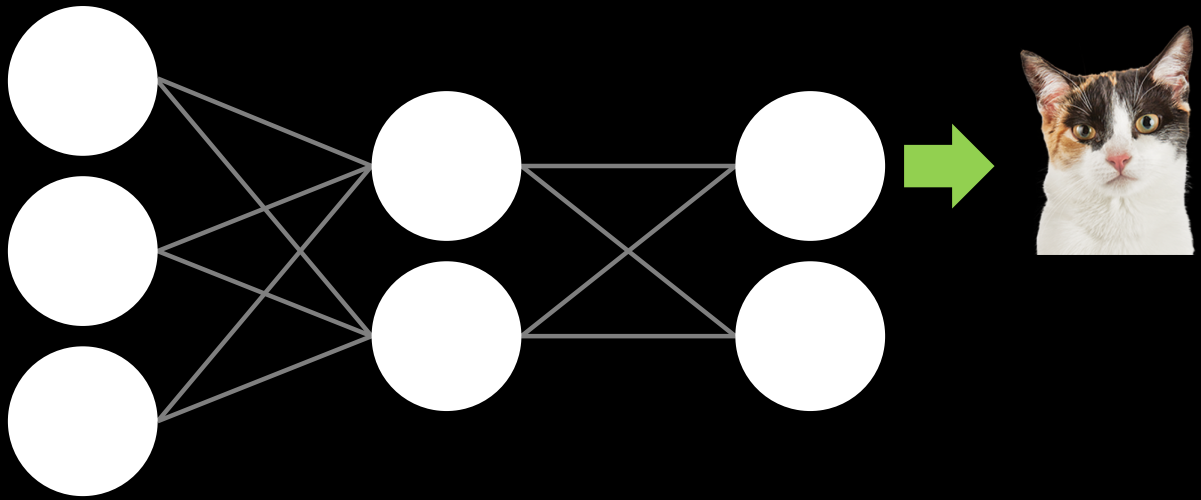
Super Boo!

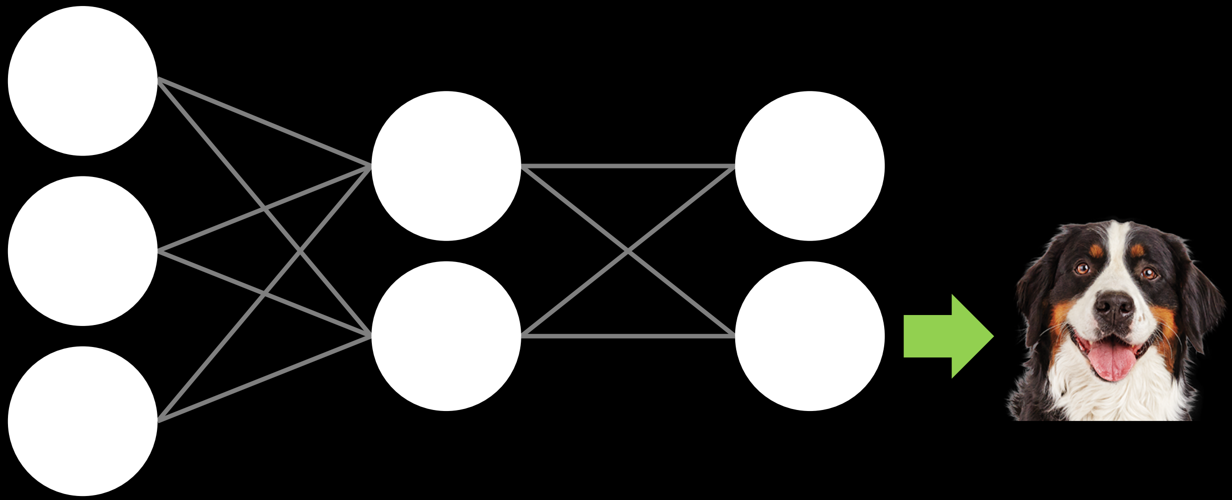
Actually, the situation is much worse, since the cost is super (13,002) high dimensional…but we proceed as if…

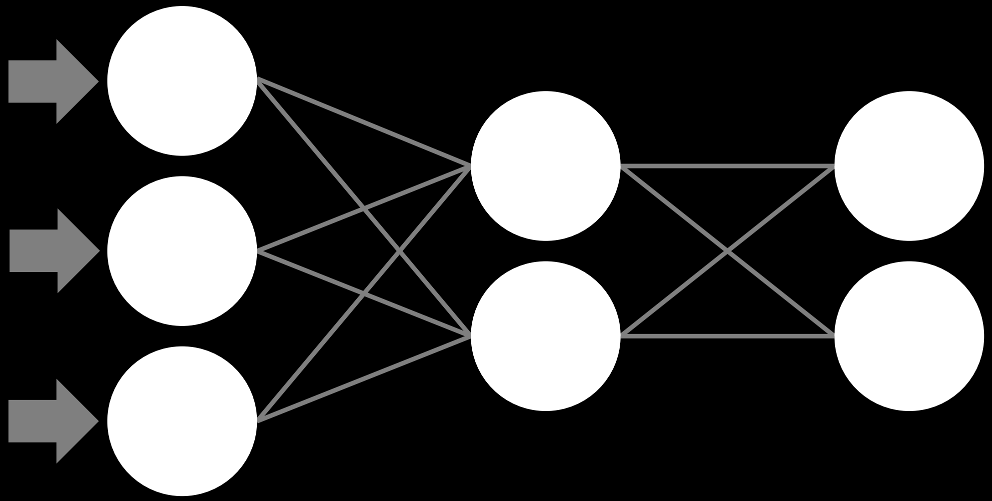


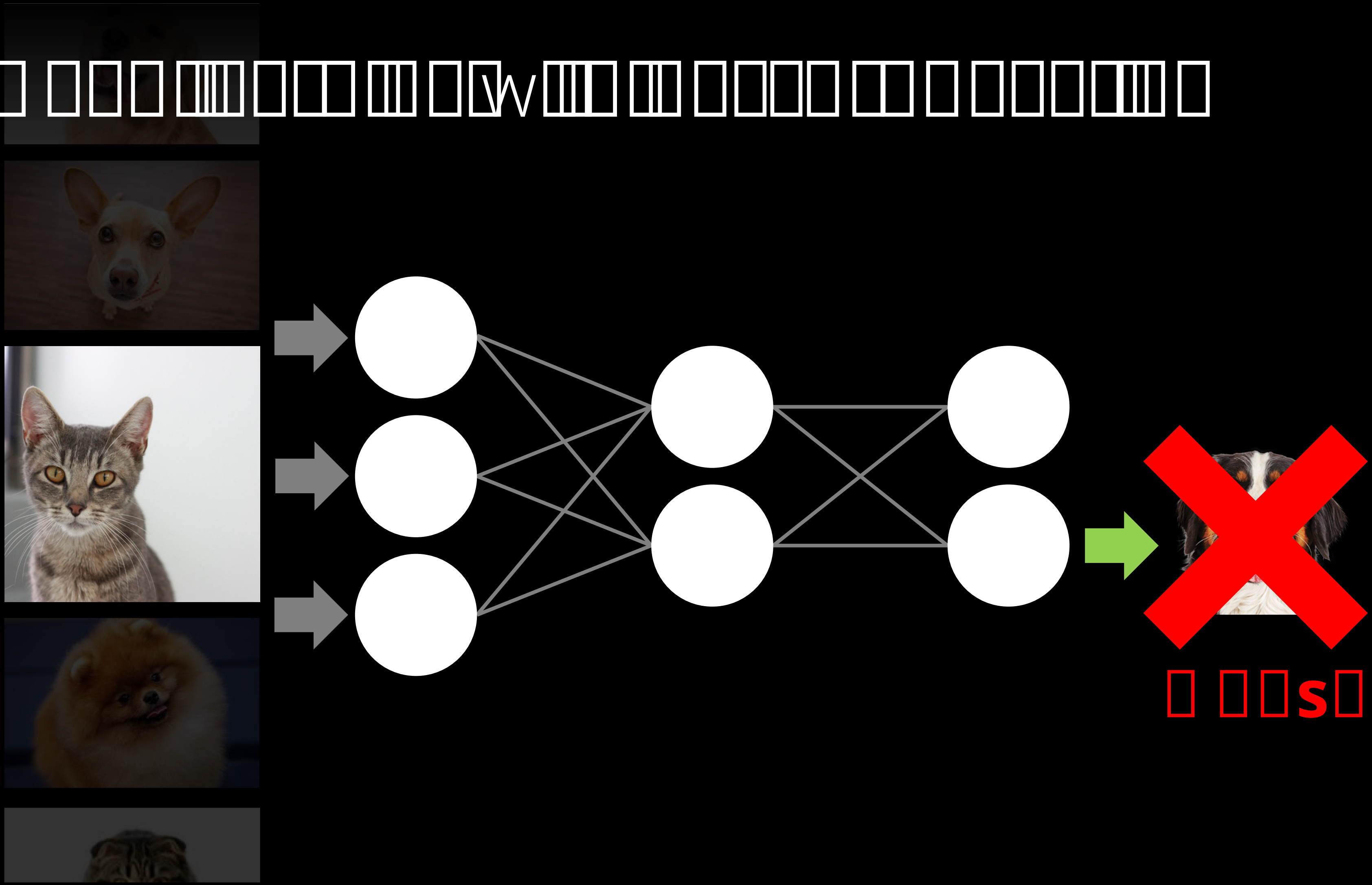


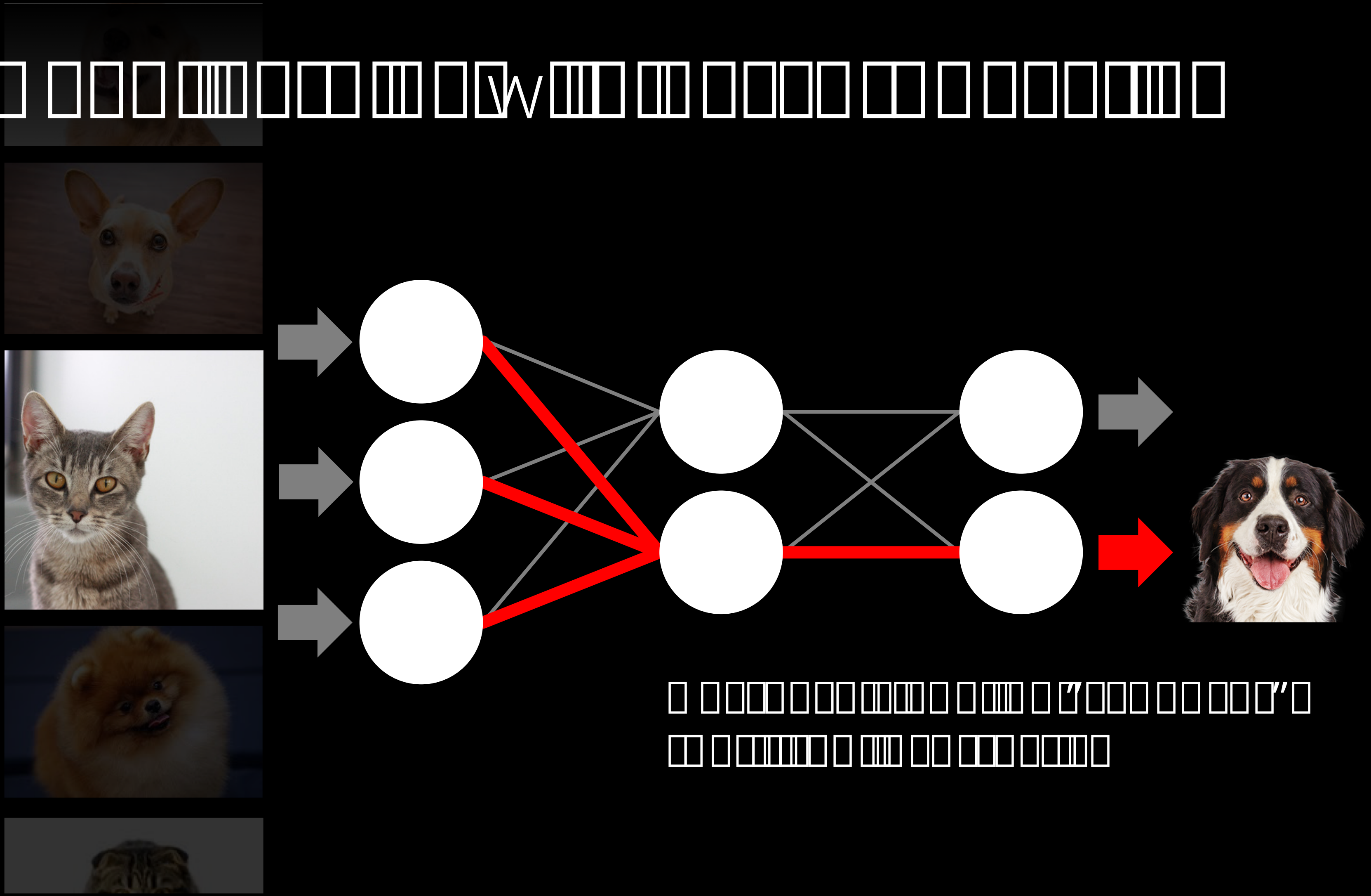


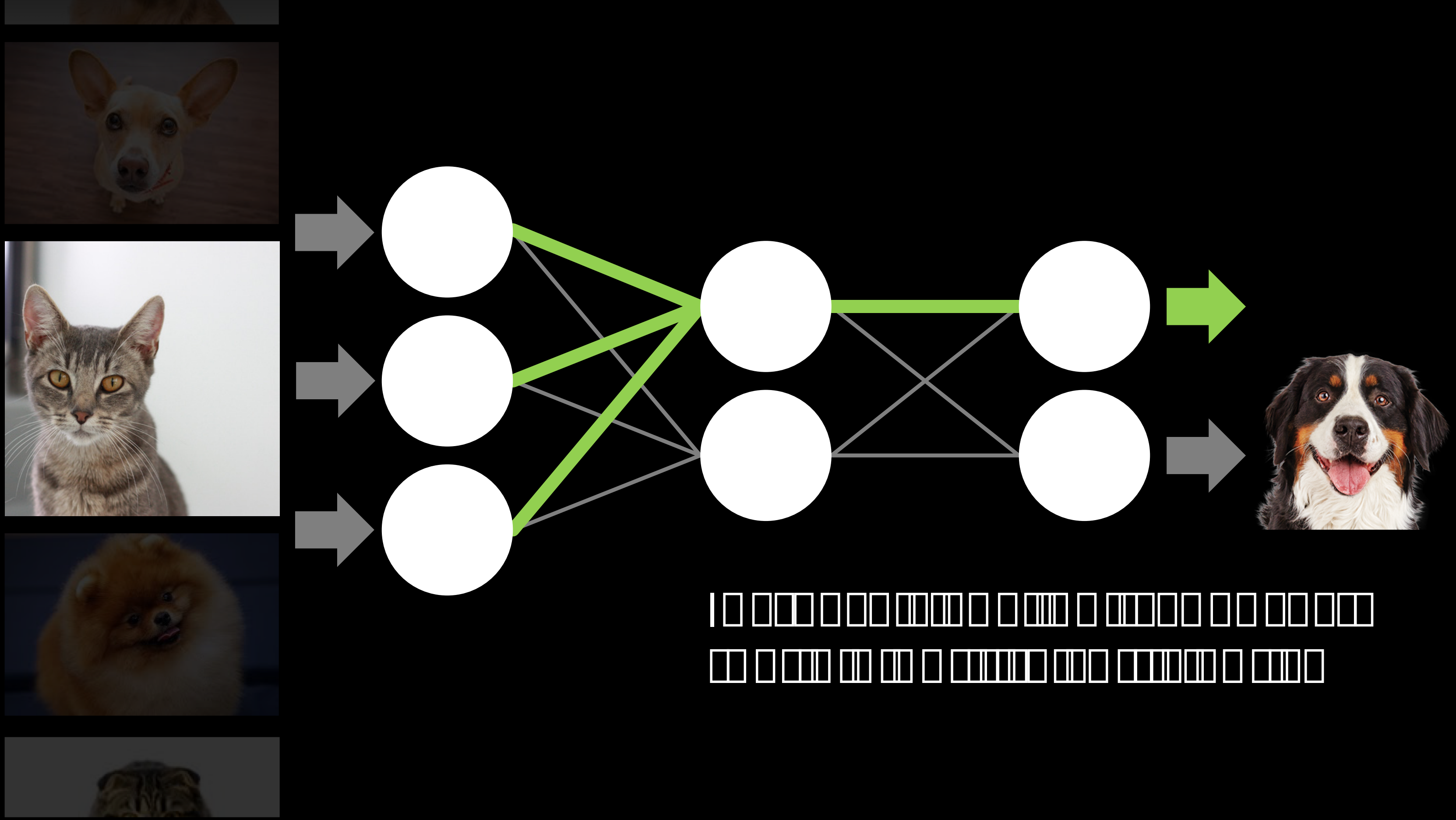












Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
initialize parameters in Θ
while not converged do
    for each training example x_i in X do
        for each θ in Θ do
            θ = θ − α (d/dθ) C_i(Θ)
        end
    end
end
```

$$\theta = \theta - \alpha \frac{d}{d\theta} C_i(\Theta)$$

Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
initialize parameters in Θ
while not converged do
    for each training example x_i in X do
        for each θ in Θ do
            θ = θ − α (d/dθ) C_i(Θ)
        end
    end
end
```

$$\text{initialize parameters in } \boxed{\Theta}$$

while *not converged* **do**

  for *each training example* $\mathbf{x}_i$ **in X do**

    for *each* θ *in* Θ **do**

      $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\Theta)$

    end

  end

end

Collection of all weights and biases in the network

Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

initialize parameters in Θ

while *not converged* **do**

  for *each training example* $\boxed{\mathbf{x}_i}$ **in** $\mathbf{X}$ **do**

    for *each* θ *in* Θ **do**

      $\theta = \theta - \alpha \frac{d}{d\theta} C_i(\Theta)$

    end

  end

end

One training example

Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
initialize parameters in Θ
while not converged do
    for each training example x_i in X do
        for each θ in Θ do
            θ = θ − α (d/dθ) C_i(Θ)
        end
    end
end
```

$$\theta = \theta - \alpha \boxed{\frac{d}{d\theta} C_i(\Theta)}$$

Partial derivative of the cost function C for each parameter (weight or bias) in the network

Algorithm 7: Stochastic gradient descent algorithm for the training of neural networks.

```
initialize parameters in Θ
while not converged do
    for each training example x_i in X do
        for each θ in Θ do
            θ = θ − α (d/dθ) C_i(Θ)
        end
    end
end
```

$$\theta = \theta - \boxed{\alpha}\frac{d}{d\theta}C_i(\Theta)$$

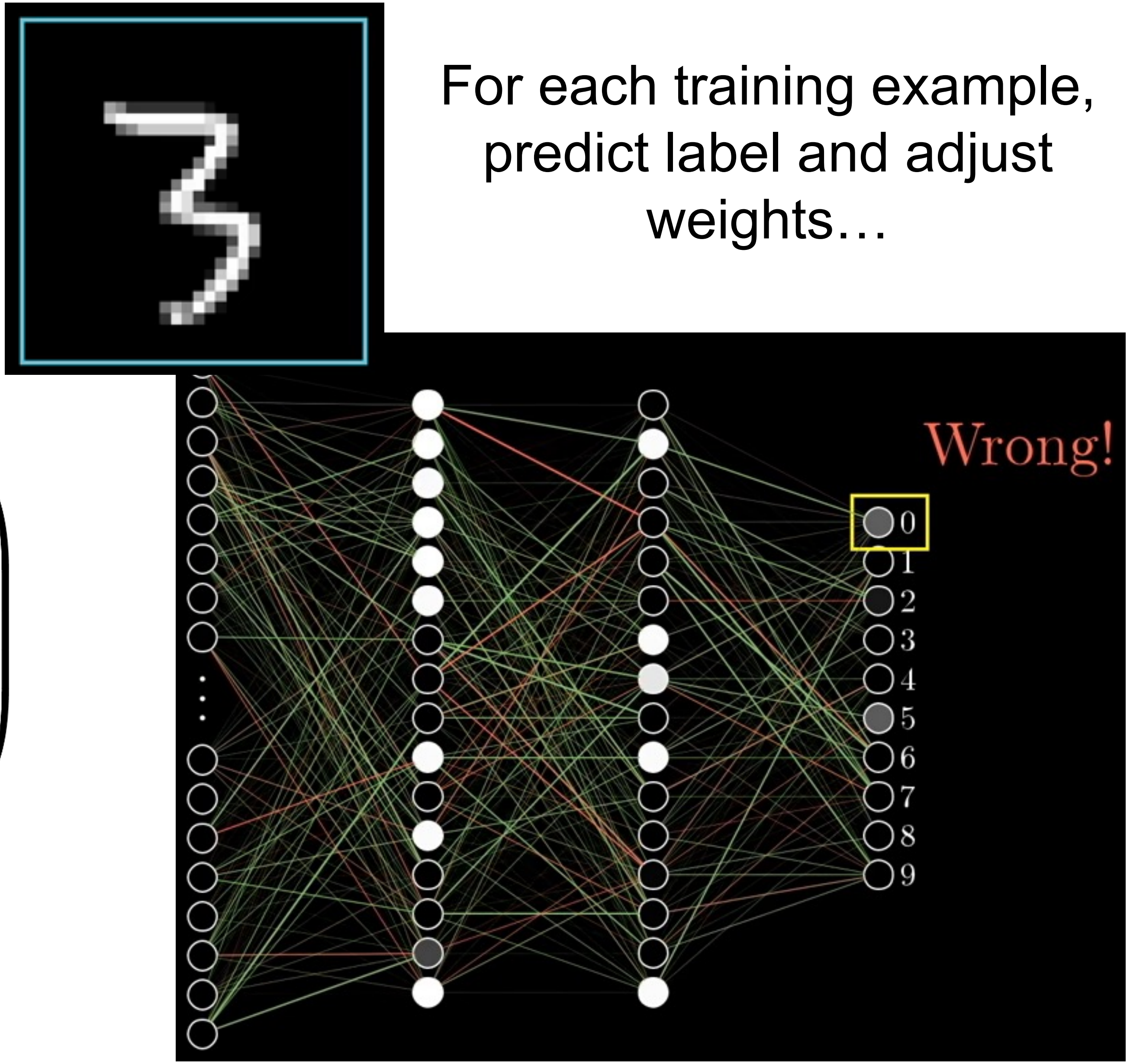
Learning rate, which is a hyper parameter

Training Multilayer Perceptron

$$X^{\text{Train}} =$$

$$Y^{\text{Train}} = \begin{pmatrix} 0 & 4 & 1 & \dots & 3 \\ 5 & 3 & 6 & \dots & 4 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 7 & 4 & 6 & \dots & 5 \end{pmatrix}$$

For each training example, predict label and adjust weights…

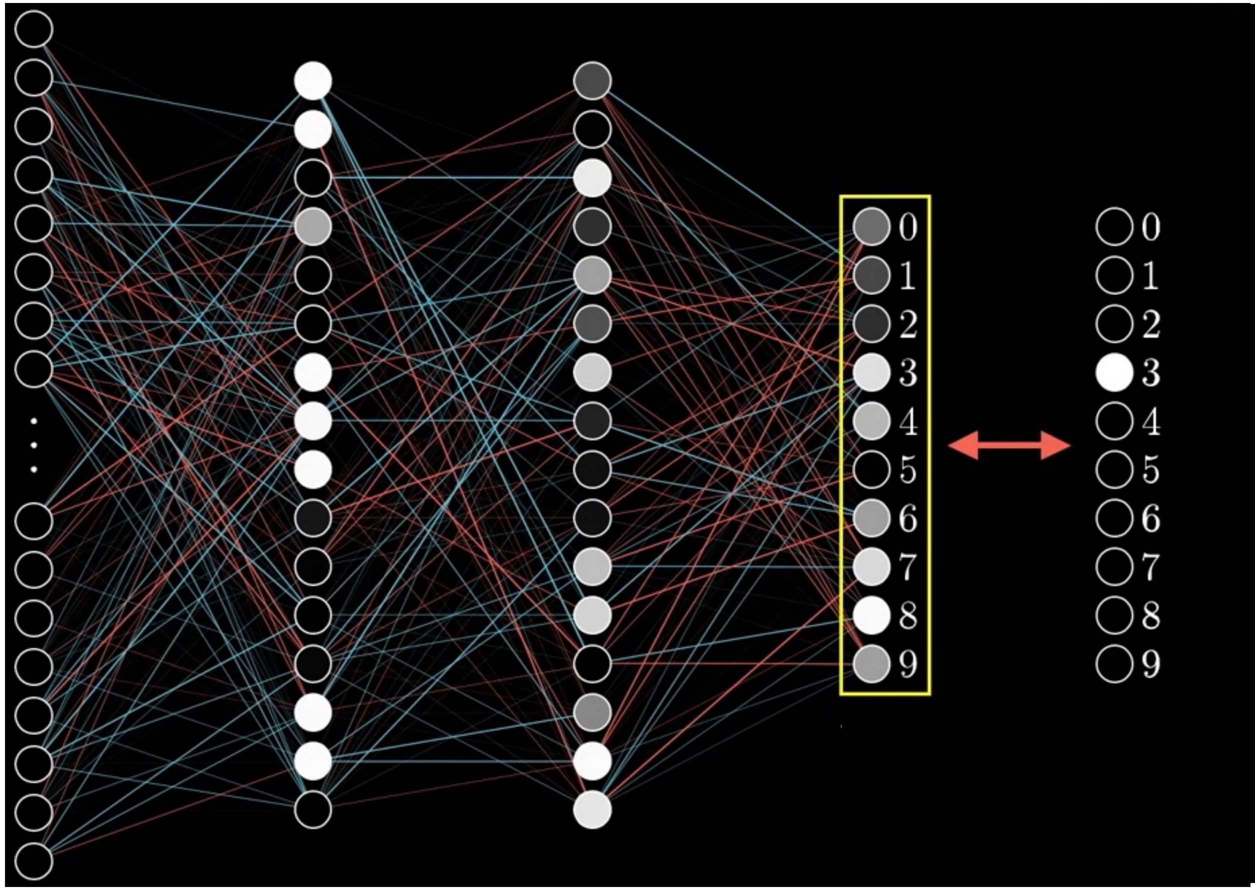


- How to score final layer output?
- How to adjust weights?

Training Multilayer Perceptron

Score based on difference between final layer and one-hot vector of true class…

Input



$$
\begin{aligned}
&(0.43 - 0.00)^2 + \\
&(0.28 - 0.00)^2 + \\
&(0.19 - 0.00)^2 + \\
&(0.88 - 1.00)^2 + \\
&(0.72 - 0.00)^2 + \\
&(0.01 - 0.00)^2 + \\
&(0.64 - 0.00)^2 + \\
&(0.86 - 0.00)^2 + \\
&(0.99 - 0.00)^2 + \\
&(0.63 - 0.00)^2
\end{aligned}
$$

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]

Computing the Derivative

So we need to compute derivatives of a super complicated function…

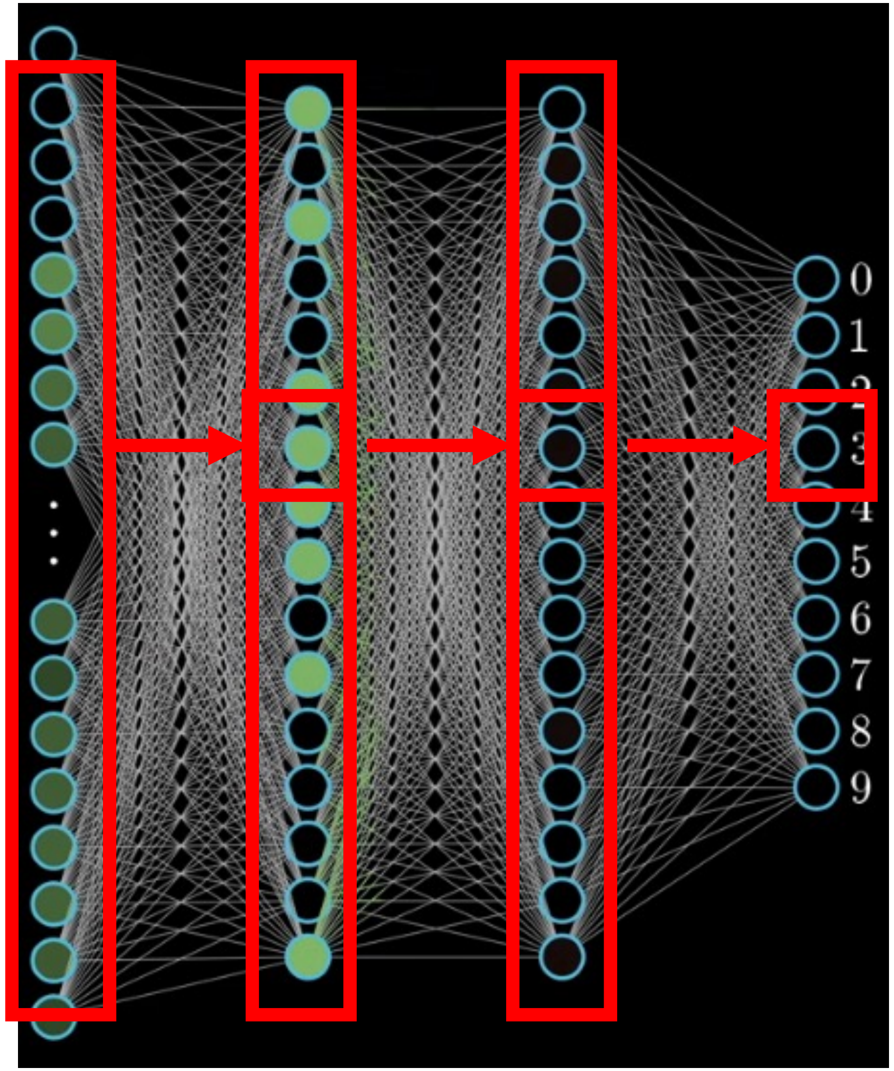
$$\frac{d}{dw}\mathcal{L}(w) = \sum_i \frac{d}{dw}\mathrm{Cost}_i(w)$$

Dropped bias terms for simplicity

- Tells us how much to turn the “tuning knob” (i.e. weight)
- But how do we compute derivatives for edge weights not directly connected to the output layer?
- Backpropagation!

Backpropagation

[Source : 3Blue1Brown : https://www.youtube.com/watch?v=aircAruvnKk]



Activation at final layer involves weighted combination of activations at previous layer…

$$\sigma(w^T x)$$

Which involves a weighted combination of the layer before it…

$$\sigma(w_n^T \sigma(w_{n-1}^T x))$$

And so on…

$$\sigma(w_n^T \sigma(w_{n-1}^T \sigma(w_{n-2}^T \sigma(\ldots))))$$

Computing the Derivative

Recall the **derivative chain rule**

$$\frac{d}{dw}f(g(w)) = \underbrace{\frac{d}{dg(w)}f(g(w))}_{\substack{\text{Derivative of f at its} \\ \text{argument g(w)} \\ \text{e.g. treat g(w) as a variable}}} \underbrace{\left(\frac{d}{dw}g(w)\right)}_{\substack{\text{Differentiate g with} \\ \text{respect to w}}}$$

Alternatively we can write this as…

$$\frac{d}{dw}f(g(w)) = f'(g(w))g'(w)$$

Derivative Chain Rule

Example Derivative of the logistic function,

$$\frac{d}{dz}\sigma(z) = \frac{d}{dz}\frac{1}{1+e^{-z}}$$

$$f(x) = \frac{1}{x}$$

$$f'(x) = -\frac{1}{x^2}$$

$$g(z) = 1 + e^{-z}$$

$$g'(z) = -e^{-z}$$

$$\begin{aligned} \sigma'(z) &= f'(g(z))g'(z) \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z)(1-\sigma(z)) \end{aligned}$$

Backpropagation

Backpropagation is the procedure of repeatedly applying the derivative chain rule to compute the full derivative

Example

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

$$\frac{d}{dz}\sigma(\sigma(z)) = \sigma(\sigma(z))(1 - \sigma(\sigma(z)))\frac{d}{dz}\sigma(z)$$

This is simply the derivative chain rule applied through the entire network, from the output to the input

Backpropagation

- Implementation-wise all we need is a function that computes the derivative of each nonlinear activation
- We can repeatedly call this function, starting at the end of the network and moving backwards
- In practice, neural network implementations use *auto differentiation* to compute the derivative on-the-fly
- Can do this efficiently on *graphical processing units (GPUs)* on extremely large training datasets

Universal Approximation Theorem

(Informally) For *any* function $f(x)$ there exists a multilayer perceptron that approximates $f(x)$ with arbitrary accuracy.

- Specific cases for arbitrary depth (number of hidden layers) and arbitrary width (number of nodes in a layer)
- Not a constructive proof (doesn't guarantee you can learn parameters)
- Corollary : The multilayer perceptron is a *universal turing machine*
- Also means it can easily overfit training data (regularization is critical)

Regularization

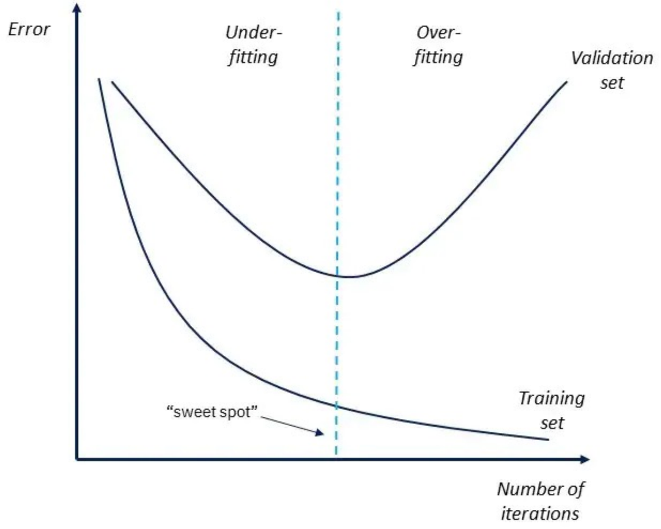
Regularization

With four parameters I can fit an elephant. With five I can make him wiggle his trunk. - John von Neumann

$$w = \arg\min_{w} \mathrm{Cost}(w) + \alpha \cdot \mathrm{Regularizer}(\mathrm{Model})$$

Our example model has 13,002 parameters…that's a lot of elephants! Regularization is critical to avoid overfitting…

…numerous regularization schemes are used in training neural networks



L2 Regularization

Formalize the regularized cost function as,

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

Consider an L2 penalty,

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top\boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

Gradient (derivative) with respect to w is given by,

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

Take a single step in the direction of the gradient,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon\left(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}}J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\right)$$

L2 Reguilarization (Weight Decay)

Written another way, a single gradient step is:

$$\boldsymbol{w} \leftarrow (1 - \underset{\uparrow}{\epsilon}\underset{\uparrow}{\alpha})\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

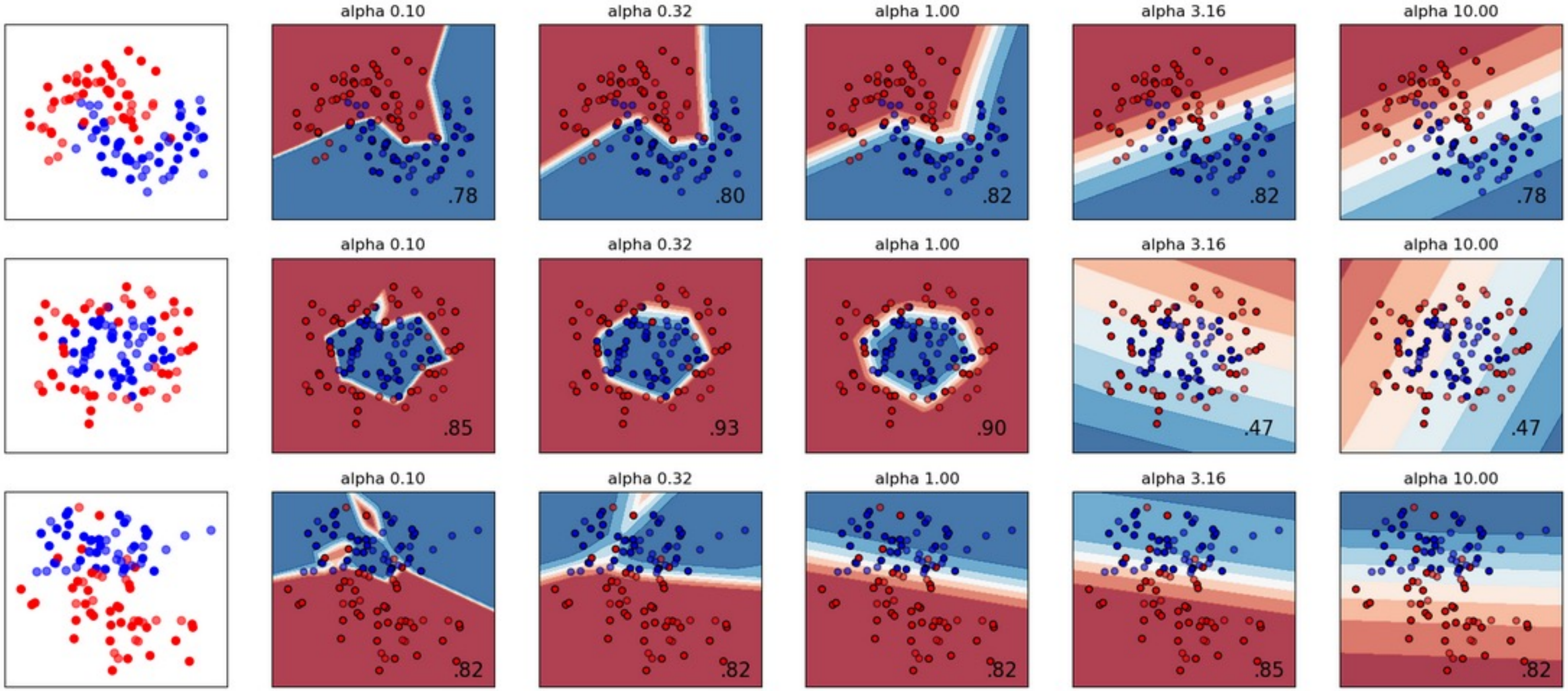
Learning Rate
(how big of a step)

Regularization
Strength (Coefficient)

- Can see this is a modification to the learning rule (gradient descent)
- "Shrinks" the weight by constant factor on each step
- Then perform usual gradient step

Regularization : Weight Decay

$$w = \arg\min_{w} \ \mathrm{Cost}(w) + \frac{\alpha}{2}\|w\|^2$$



L1 Regularization

$$\widetilde{J}(w) = J(w) + \alpha\|w\|_1$$

(Sub-)gradient given by,

$$\nabla_{\boldsymbol{w}}\tilde{J}(\boldsymbol{w};\boldsymbol{X},\boldsymbol{y}) = \alpha\text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}}J(\boldsymbol{X},\boldsymbol{y};\boldsymbol{w})$$

- Very different effect from L2 weight decay
- Regularization contribution no longer scales linearly with each w
- Constant addition with sign equal to sign(w)
- Has a *sparsity-inducing property* (forces some weights to w=0)

L1 Regularization

$$w_i = \operatorname{sign}(w_i^*) \max\left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}$$

Consider the case where $w_i^* > 0$ for all i. There are two possible cases,

$w_i^* \leq \frac{\alpha}{H_{i,i}}$:

- Optimal value is just w_i=0
- Contribution of J(w;X,y) is "overwhelmed" by L1 regularizer

$w_i^* > \frac{\alpha}{H_{i,i}}$:

- Shifts w_i in the direction of 0 by distance equal to a/H

Similar process for w<0 but in opposite direction.

Sparse Representations

L1 regularization induces **sparse parameterization** – many parameters 0

Representational sparsity enforces many data elements 0 (or close to it)

Accomplished by same set of mechanisms as sparse param – norm penalty on representation

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{h})$$

e.g. L1 penalty

Sparse Parameterization

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix}} = \underset{\boldsymbol{A} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix}} \underset{\boldsymbol{x} \in \mathbb{R}^n}{\begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix}}$$

$$\underset{\boldsymbol{y} \in \mathbb{R}^m}{\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix}} = \underset{\boldsymbol{B} \in \mathbb{R}^{m \times n}}{\begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix}} \underset{\boldsymbol{h} \in \mathbb{R}^n}{\begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix}}$$

Sparse Representation

Parameter Tying / Sharing

- Introduces inductive bias
  - There should be dependencies among parameters
  - Parameters should be close / similar
- Can use previously-trained model on similar task
- Parameter norm penalty is one way
- Hard constraints force sets of parameters to be equal
  - Known as *parameter sharing*
  - Only subset of unique parameters needs to be stored in memory

(Goodfellow 2016)

Dataset Augmentation

- Train on more data (always more data)
- What if we don’t have more data? (Make up more)
- Easiest for classification
- Generate new (x,y) pairs by transforming x in dataset for each y
- Not readily applicable to many other tasks
  - E.g. hard for density estimation unles we’ve solved the density estimation prob.
- Particularly effective for object recognition
  - Translation
  - Scaling
  - Rotation
  - …

(Goodfellow 2016)

Dataset Augmentation

Affine Distortion

Noise

Elastic Deformation

Horizontal flip

Random Translation

Hue Shift

(Goodfellow 2016)

Dataset Augmentation

- Need to avoid transformations that change class
- For example mirror “b” to produce “d”
- Rotation turns “6” into “9”
- Some transformations are not easy to perform, e.g. out-of-plane rotation

(Goodfellow 2016)

Label Smoothing

- Many datasets have some mistakes in labels y
- Inject noise in labels at output
  - Assume label is correct with probability 1-e (for some small e)
  - Otherwise any other label is assigned
- Can incorporate this into cost function analytically
- Label smoothing regularizes model based on softmax
  - Replaces hard assignment with 1-e and e/(k-1) ; for k labels
  - Can use standard cross-entropy loss with soft targets

(Goodfellow 2016)

Learning Curves – Early Stopping

Early stopping: terminate while validation set performance is better

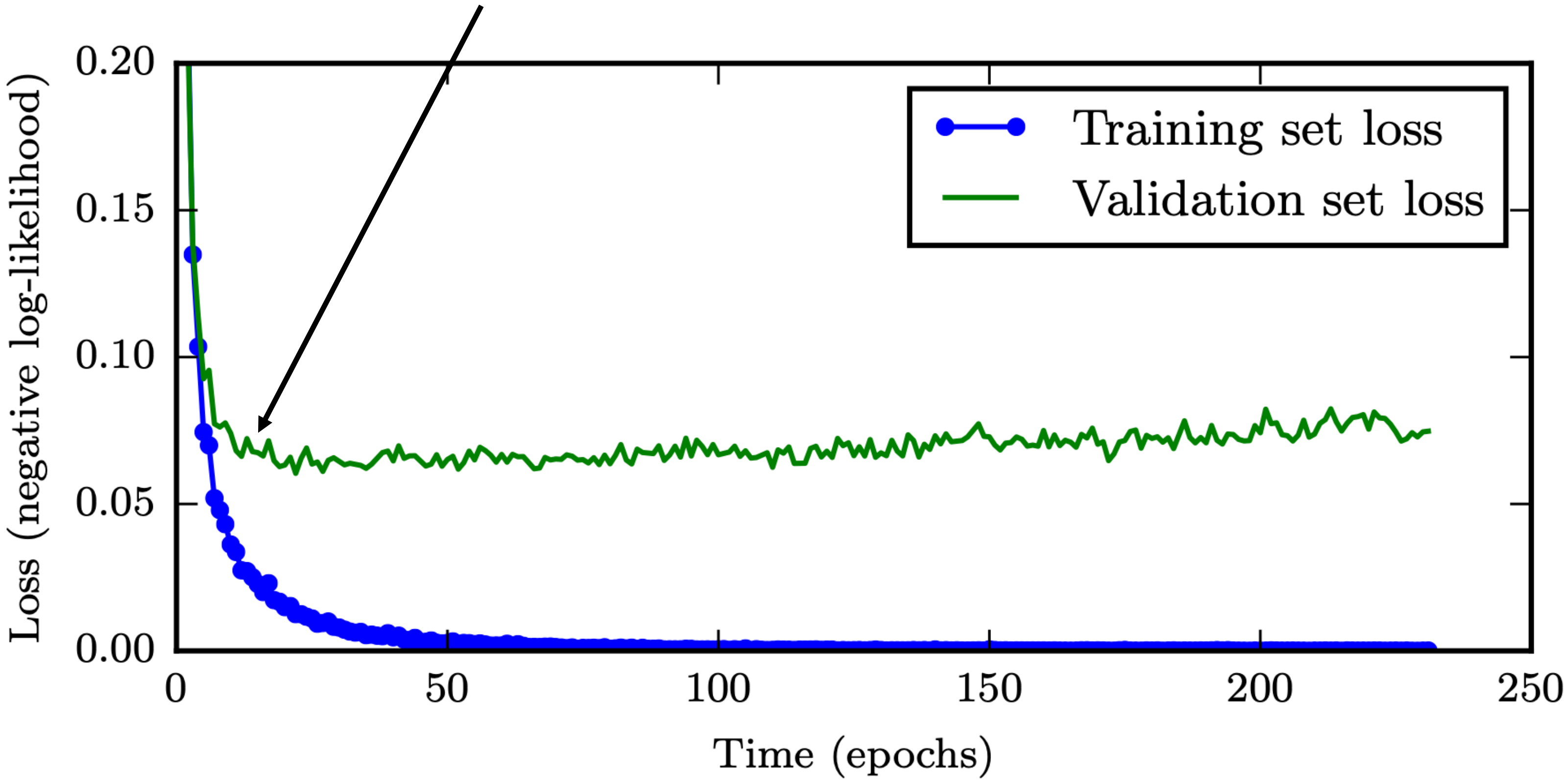


Figure 7.3

(Goodfellow 2016)

Algorithm 7.1 The early stopping meta-algorithm for determining the best amount of time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_o$
$i \leftarrow 0$
$j \leftarrow 0$
$v \leftarrow \infty$
$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
$i^* \leftarrow i$
while $j < p$ **do**
  Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
  $i \leftarrow i + n$
  $v' \leftarrow \text{ValidationSetError}(\boldsymbol{\theta})$
  if $v' < v$ **then**
    $j \leftarrow 0$
    $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
    $i^* \leftarrow i$
    $v \leftarrow v'$
  else
    $j \leftarrow j + 1$
  end if
end while
Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*.

(Goodfellow 2016)

Early Stopping

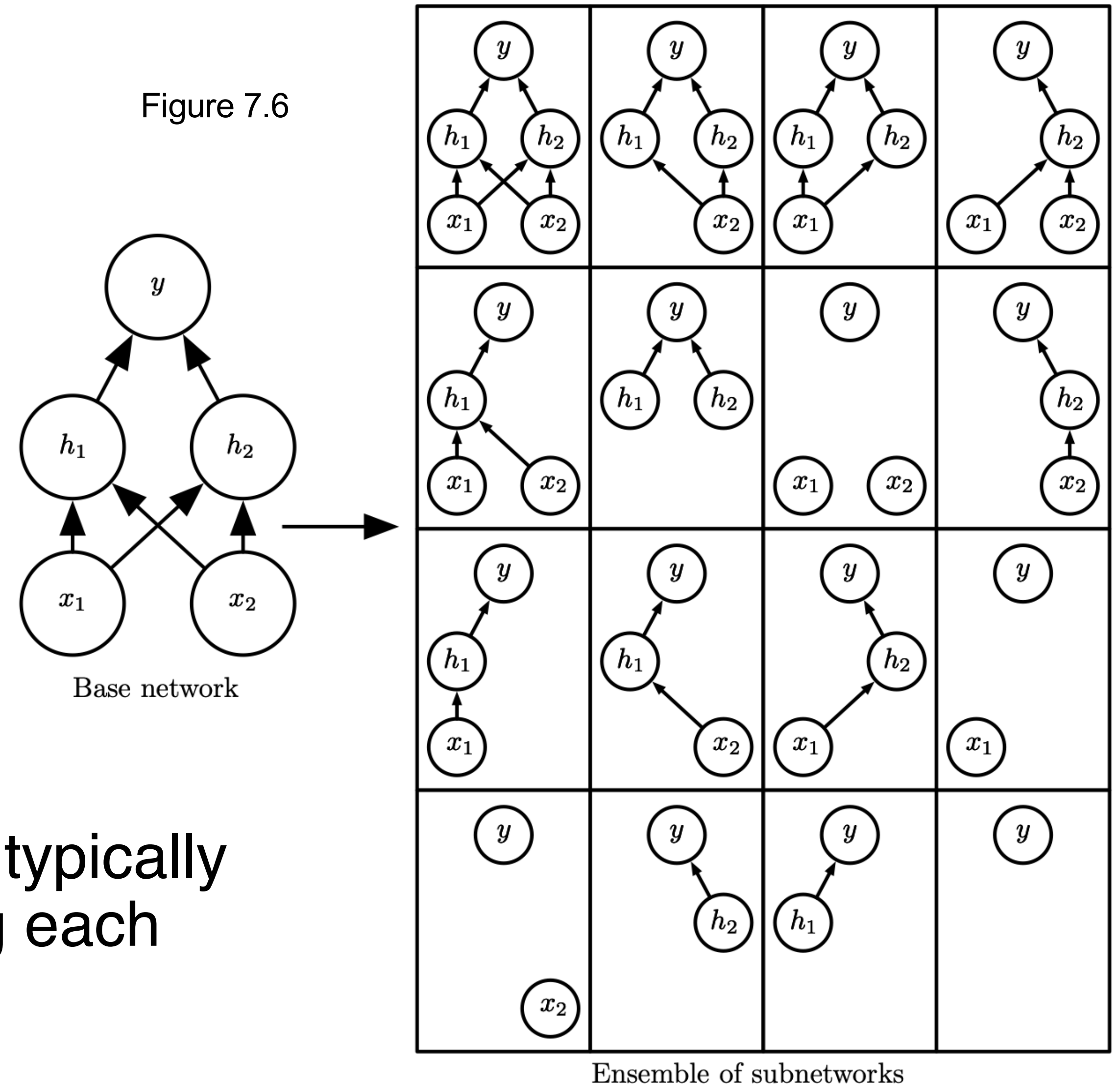
- Think of it as efficient hyperparameter selection algorithm (number of training steps)
- Requires almost no change to underlying training procedure
  - Contrast with weight decay that requires hyperparameter tuning
- Can be used alone or in conjunction with other regularization
- Can conclude with a training stage that includes all training data
  - Initialize model and retrain for same number of steps
    - Same number of parameter updates or epochs?
  - Continue from current parameters
    - How many training steps?
    - Periodically check validation set (which is now part of training)

(Goodfellow 2016)

Dropout

Provides ensemble of exponentially many ANNs - all subnetworks formed by removing subset of edges / nodes

Figure 7.6



Each time we load a minibatch, randomly remove set of edges / nodes

Includes input and hidden nodes – typically different probabilities of dropping each

(Goodfellow 2016)

Dropout

- Srivastava et al. (2014) showed more effective than weight decay and other “simple” regularization methods

- Computationally very cheap; O(n) computation per example per update
- Doesn’t significantly limit type of model that can be used

- Can slow training and require larger model sizes

- Less effective when very few training examples available

- ”Fast Dropout” – Don’t stochastically drop edges; estimate average

(Goodfellow 2016)

Regularization

- L1+L2 (elastic net) regularization
- **Dropout** Each iteration randomly selects a small number of edges to temporarily exclude from the network (weights=0)
- **Data Augmentation** Synthetically expand training data by applying random transformations
- **Early stopping** Just as it sounds…stop the network before reaching a local minimum…dumb-but-effective

Example

Play with a small multilayer perceptron on a binary classification task…

https://playground.tensorflow.org/

sklearn.neural_network.MLPClassifier

hidden_layer_sizes : *tuple, length = n_layers - 2, default=(100,)*

The ith element represents the number of neurons in the ith hidden layer.

activation : *{'identity', 'logistic', 'tanh', 'relu'}, default='relu'*

Activation function for the hidden layer.

solver : *{'lbfgs', 'sgd', 'adam'}, default='adam'*

The solver for weight optimization.

alpha : *float, default=0.0001*

L2 penalty (regularization term) parameter.

learning_rate : *{'constant', 'invscaling', 'adaptive'}, default='constant'*

Learning rate schedule for weight updates.

early_stopping : *bool, default=False*

Whether to use early stopping to terminate training when validation score is not improving. If set to true,

Scikit-Learn : Multilayer Perceptron

Fetch MNIST data from www.openml.org :

```
X, y = fetch_openml("mnist_784", version=1, return_X_y=True)
X = X / 255.0
```

Train test split (60k / 10k),

```
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]
```

Create MLP classifier instance,

- Single hidden layer (50 nodes)
- Use stochastic gradient descent
- Maximum of 10 learning iterations
- Small L2 regularization alpha=1e-4

```
mlp = MLPClassifier(
    hidden_layer_sizes=(50,),
    max_iter=10,
    alpha=1e-4,
    solver="sgd",
    verbose=10,
    random_state=1,
    learning_rate_init=0.1,
)
```

Scikit-Learn : Multilayer Perceptron

Fit the MLP and print stuff…

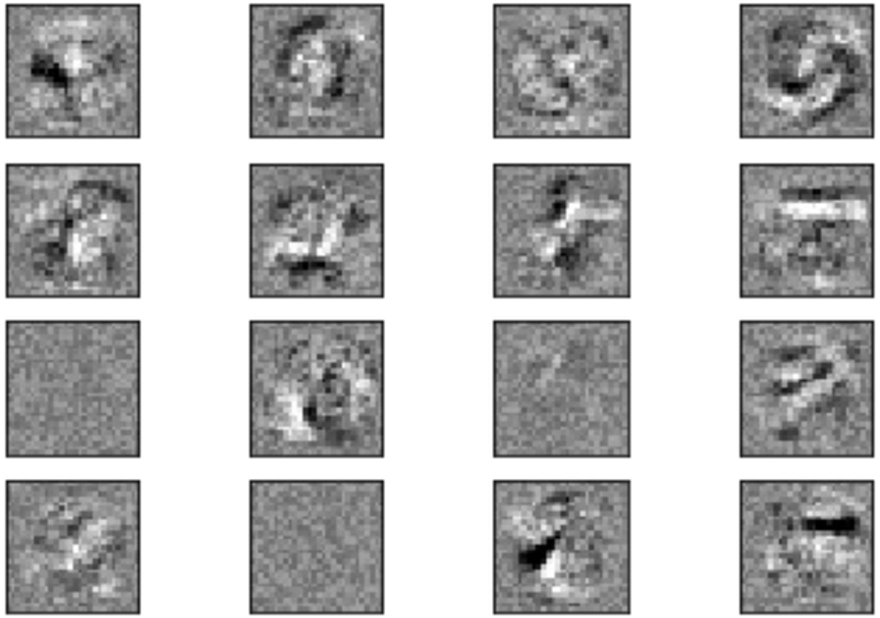
```
mlp.fit(X_train, y_train)
```

```
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

Visualize the weights for each node…

```
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray,
               vmin=0.5 * vmin, vmax=0.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())
```

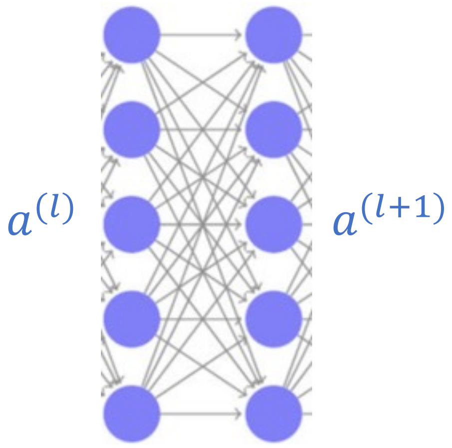


…magnitude of weights indicates which input features are important in prediction

Convolutional Neural Networks

NNs for images

- Fully-connected (FC) layers do not scale well to images (width x height x #channels)
  - Need for smaller number of parameters
- Note: FCs can learn (pattern, location) combinations in images
  - The learned patterns do not generalize to different spatial locations.

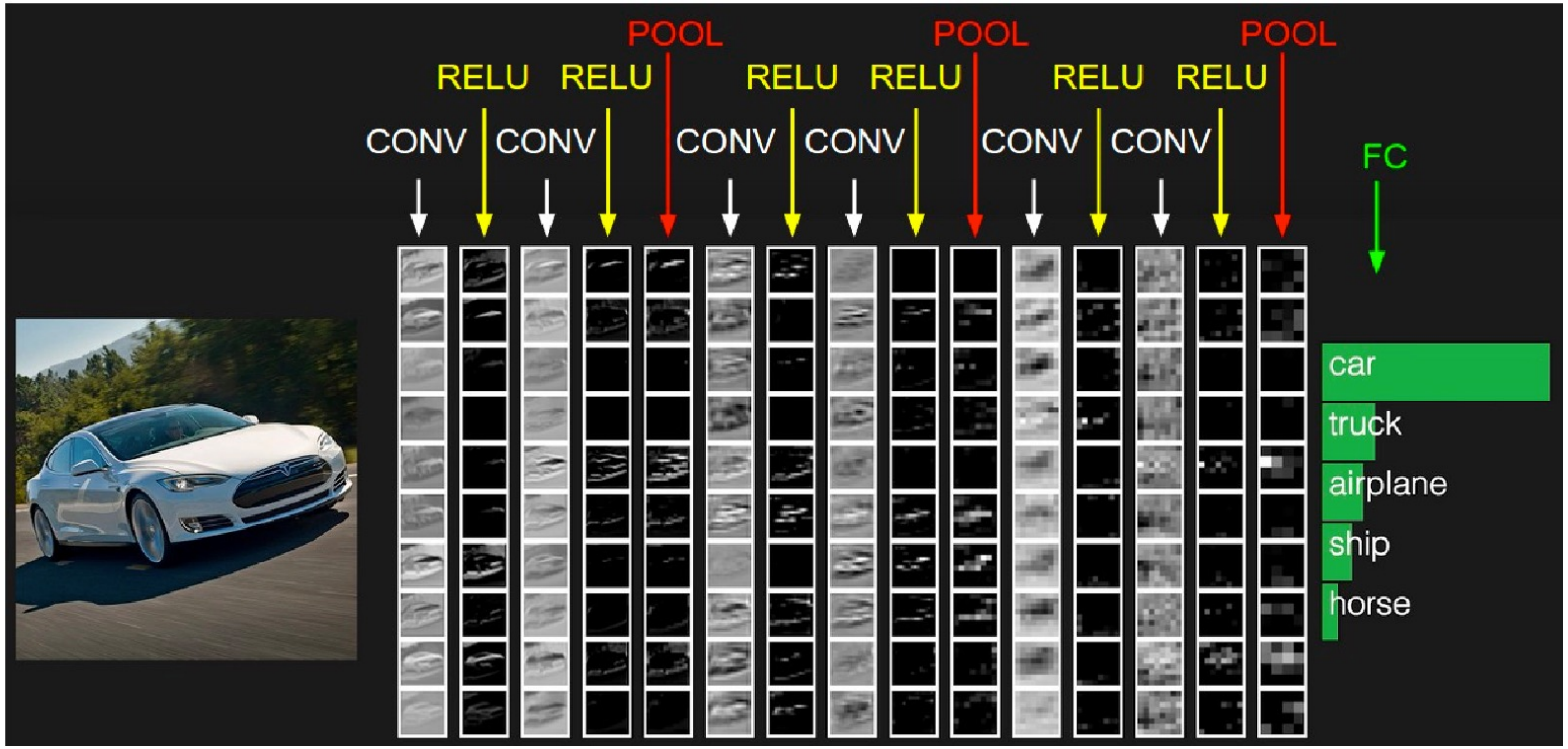


- Can we capture local patterns (e.g. *existence* of a wheel in an image) regardless of the spatial location in the image and leverage them for better classification?
  - low level: edge of some orientation, a patch of some color
  - high level: shape of a wheel
  - i.e. can we learn a group of neurons that detect patterns at all locations?

- Encodes inductive bias

Convolutional neural networks (CNN)

- A.K.A. ConvNet architecture
- A set of neural network architecture that consists of
  - convolutional layers
  - pooling layers
  - fully-connected (FC) layers

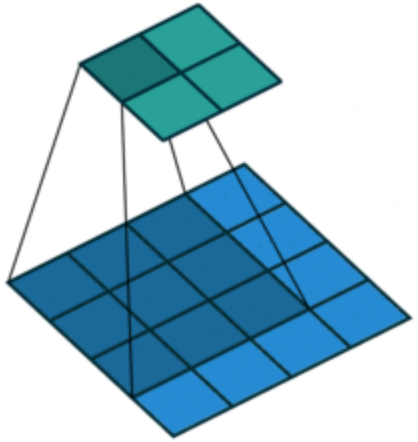


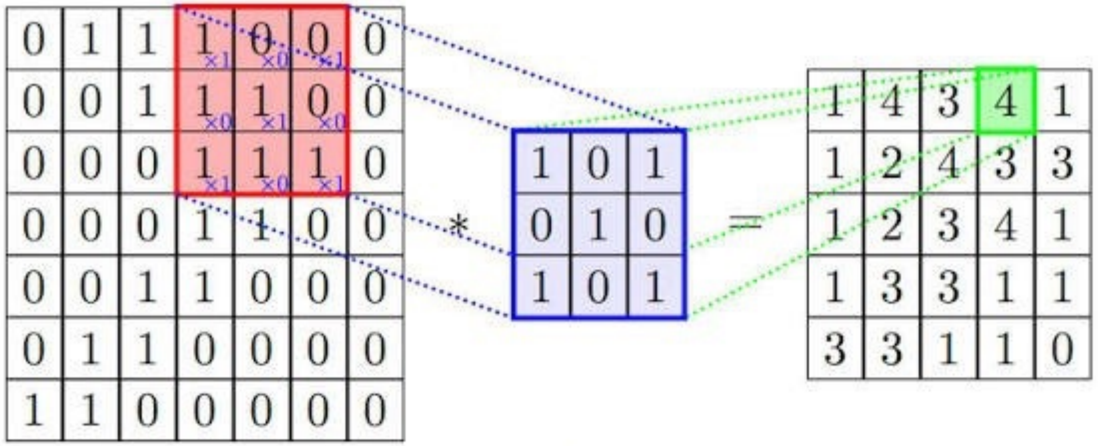
(Stanford CS231n)

Convolution for single-channel images

Consider one filter with weights $\{w_{i,j}\}$ with size F x F

- For every F x F region of the image, perform inner product (= element wise product, then sum them all)
- Q: given a w x h image, after convolution with a F x F filter, what is the size of the resulting image?
- Terminologies: filter size, receptive field size, kernel.

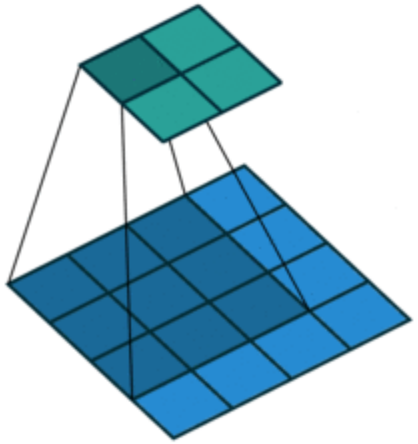




Vincent Dumoulin, Francesco Visin - A guide to convolution arithmetic for deep learning

Convolution: Some Intuition

Define the convolution of filter f on image I as:



$$(I * f)(x) = \sum_m \sum_n f(x-m, y-n) I(m,n)$$

Many ML libraries *actually* implement cross-correlation:

$$(f * I)(x) = \sum_m \sum_n f(x,y) I(x+m, y+n)$$

Learning finds good values for the convolution filter…

Convolutional layer for multi-channel images

Input: w (width) x h (height) x c (#channels)

- E.g. 32 x 32 x 3
- 3 channels: R, G, and B

A convolutional filter on such image is of shape F x F x c

- Only spatial structure in the first two dimensions
- Denoted by $\{w_{i,j,k}\}$

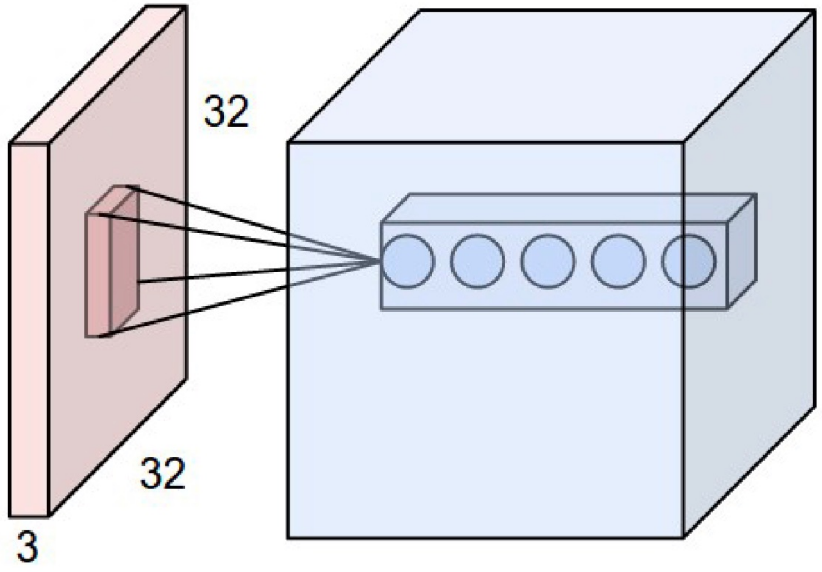
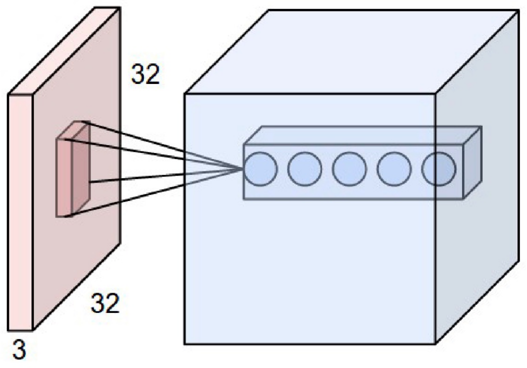
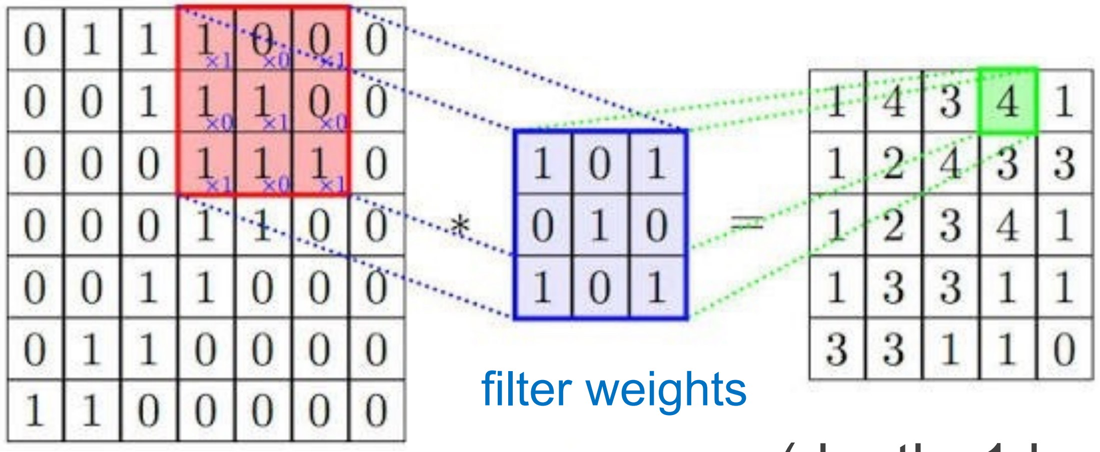


image from Stanford CS231n

Convolutional layer: visual explanation

- Consider one filter with weights $\{w_{i,j,k}\}$ with 5 x 5 x 3
  - Imagine a sliding 3D window.
  - **Convolution:** For every 5 x 5 region of the image, perform inner product (= element wise product, then sum them all)
  - Then apply the activation function (e.g., ReLU)
- Results in 28 x 28 x 1 – called **activation map**.
- Now, we can do K of these filters but with different weights $\{w_{i,j,k}^{(\ell)}\}$ for $\ell \in [K]$ => output is 28 x 28 x K





(depth=1 here)

(image from https://www.quora.com/Why-do-we-use-convolutional-layers)

Convolutional Layer: Why is it useful?

The set of weights represent a pattern (i.e., diagonal edge). The activation map represents 'where the pattern has occurred'.

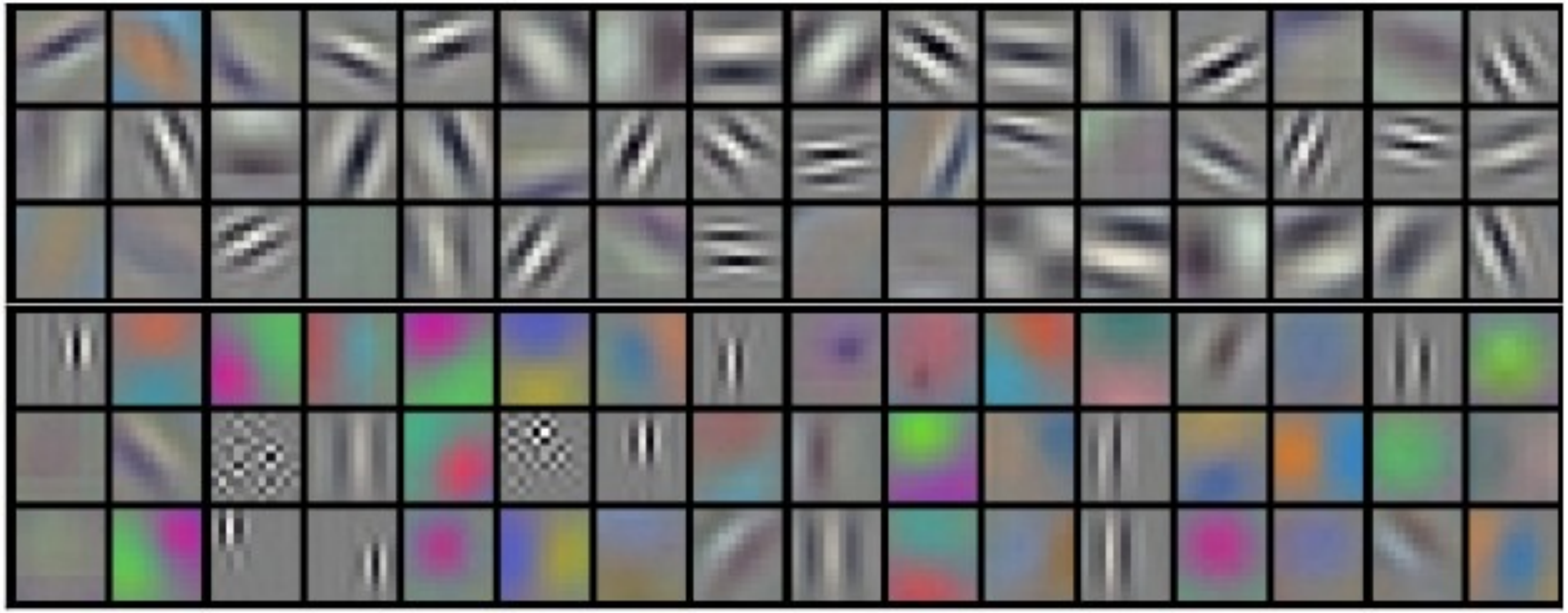


image from Stanford CS231n

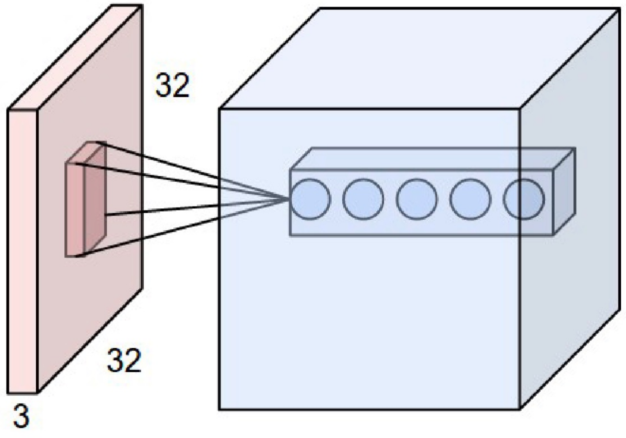
Convolutional Layers Beyond the First Layer

Generalization: conv layer as the 2nd layer or more

- Input **volume** (3d object with size w x h x d):
  - the d (called depth) is not necessarily 3
- Output **volume:** size w’ x h’ x d’, where d’ is the number of filters at the current layer.

Interpretation: patterns over the patterns.

- Each filter now convolves and combines d’ activation maps for each spatial location.
- e.g., combinations of particular edges and textures



Convolutional Layer: More Details

Stride length S

- Skip input regions; Move the sliding window of a filter not by 1 but by S.
- E.g., S=2 means skipping every other 5 by 5 region.

Zero-padding P: add P number of artificial pixels with value 0 around the input image on both sides

- To ensure the spatial dimension is maintained (otherwise, patterns at the corners are not detected well)
- If we use P=1, then the activation map will be 30 x 30, not 28 x 28 in our example!

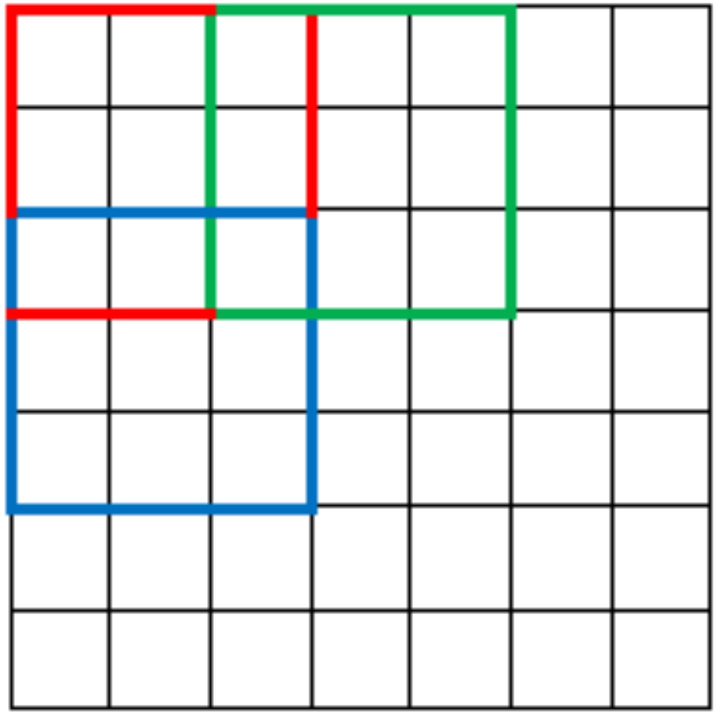


image from https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Example

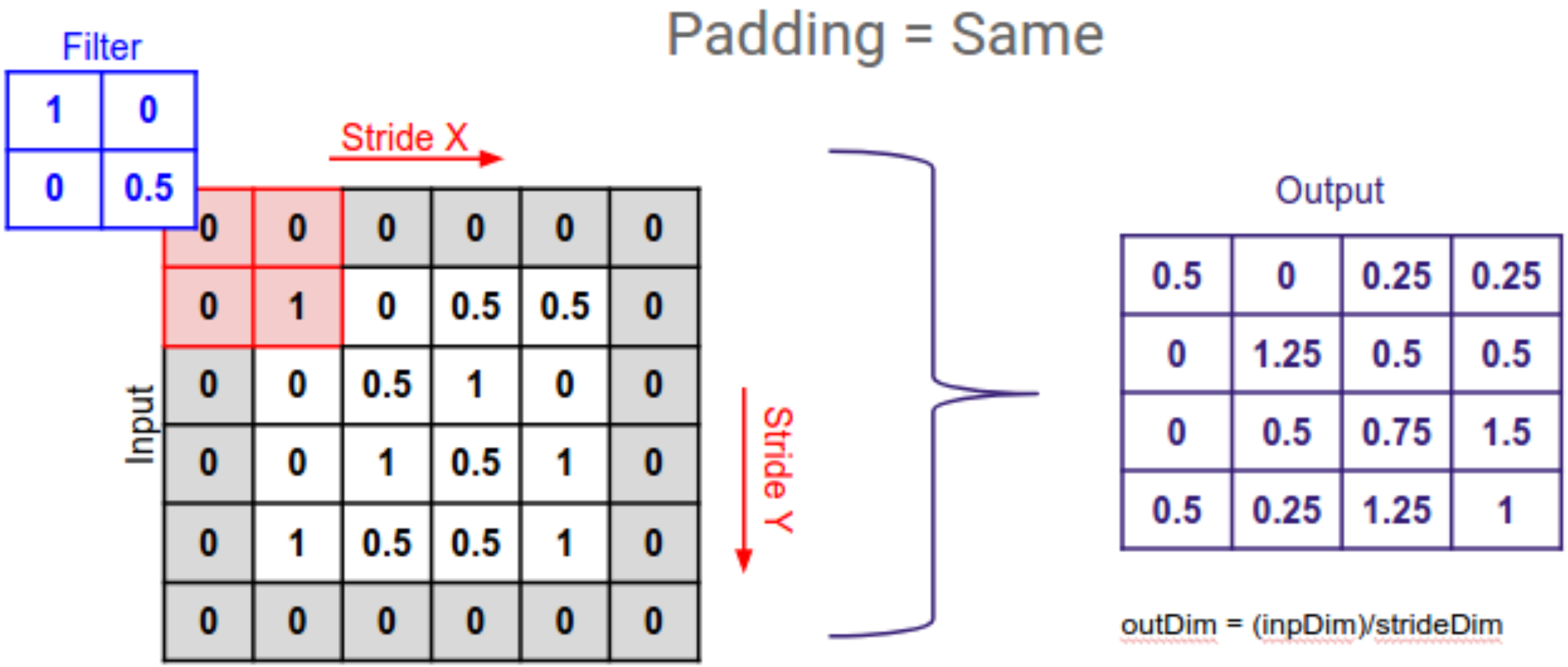
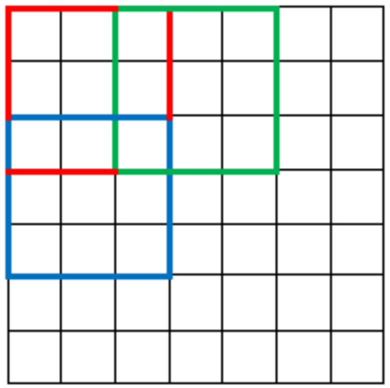


image from https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7

Convolutional Layer: More Details



Stride length S

- Skip input regions; Move the sliding window of a filter not by 1 but by S.
- E.g., S=2 means skipping every other 5 by 5 region.

Zero-padding P: add P number of artificial pixels with value 0 input image.

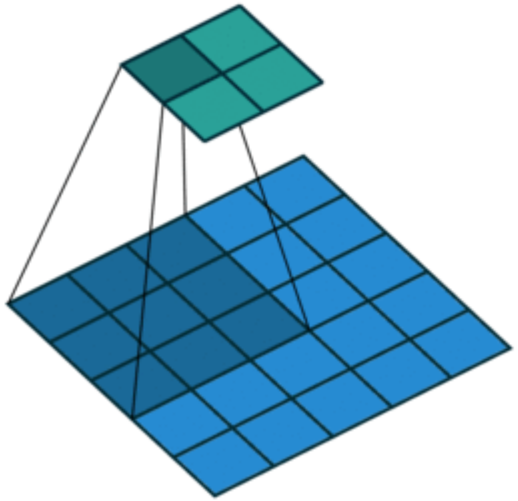
- To ensure the spatial dimension is maintained (otherwise, patterns at the corners are not detected well)
- If we use P=2, then the activation map will be 32 by 32 not 28 by 28 in our example!

Rules (same goes for height)

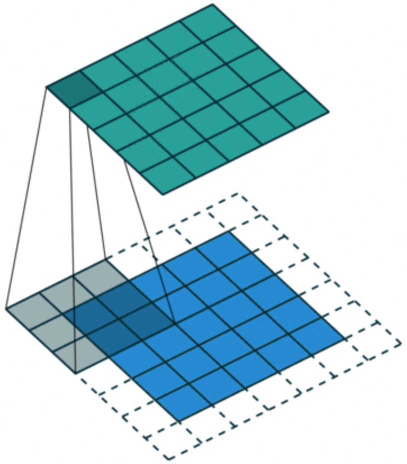
- W: input volume width, F: filter width (usually, the filter has the same width and height)
- The output width K = floor((W – F + 2P)/S) + 1
- E.g., W=32, F=5, P=0, S=1 => K = 28
- E.g., W=32, F=5, P=2, S=1 => K = 32

Strides and padding: animations

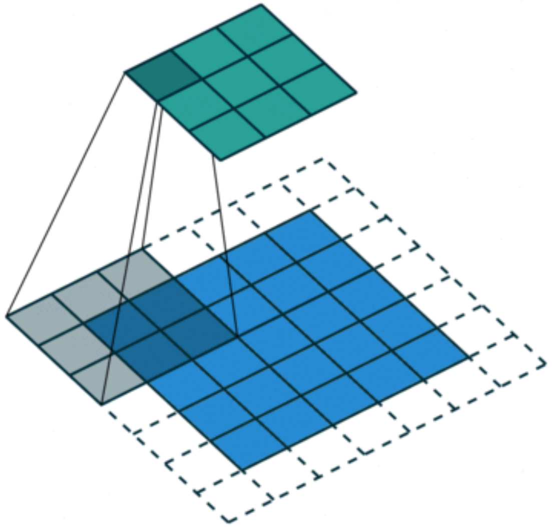
Strides only



Padding only



Strides + Padding



Vincent Dumoulin, Francesco Visin - [A guide to convolution arithmetic for deep learning]()

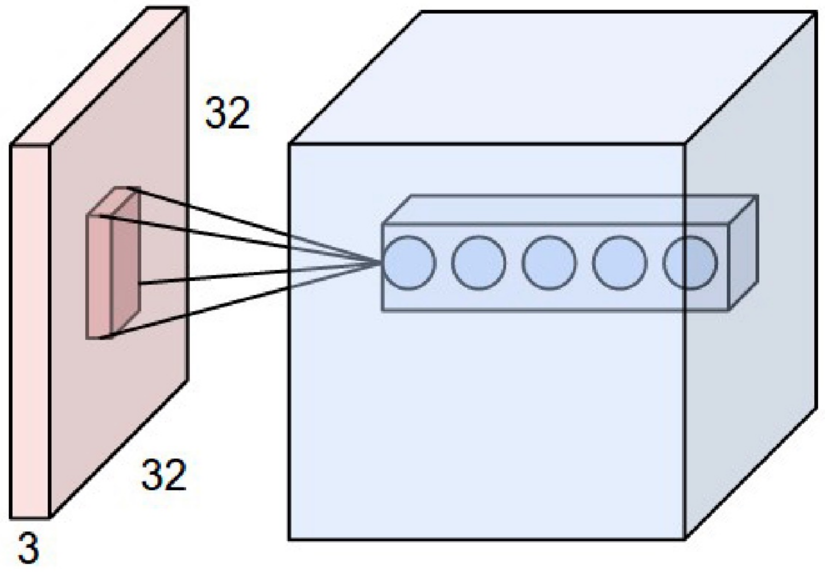
Convolutional Layer: Summary

Input $W_1 \times H_1 \times D_1$ (width, height, depth)

Hyperparameters # of filters K, filter size (=width=height) F, stride S, zero-padding P

Output $W_2 \times H_2 \times D_2$

$$W_2 = \left\lfloor \frac{W_1 - F + 2P}{S} \right\rfloor + 1,$$
$$H_2 = \left\lfloor \frac{H_1 - F + 2P}{S} \right\rfloor + 1,$$
$$D_2 = K$$



More terminology: <u>depth slice</u> (W by H by 1), <u>depth column</u> (1 by 1 by D)

Comparison: FC vs Conv

- Conv layer allows *parsimonious* representations:
  - Inter-layer connections are **local**
  - **parameter is shared** across spatial locations.

- In AlexNet, input is 227 by 227 by 3, and the first conv layer output is 55 by 55 by 96 (96 filters)
  - Each filter has 11*11*3 weights with 1 bias => 364 parameters
  - 364*96 = <u>34,944</u> total parameters are used to compute the output 55*55*96 = <u>290,400</u>

- What if we didn't do **parameter sharing**? I.e., for each region of image, use independent filter parameter w.
  - roughly, 290,400 * 364 = 105,705,600

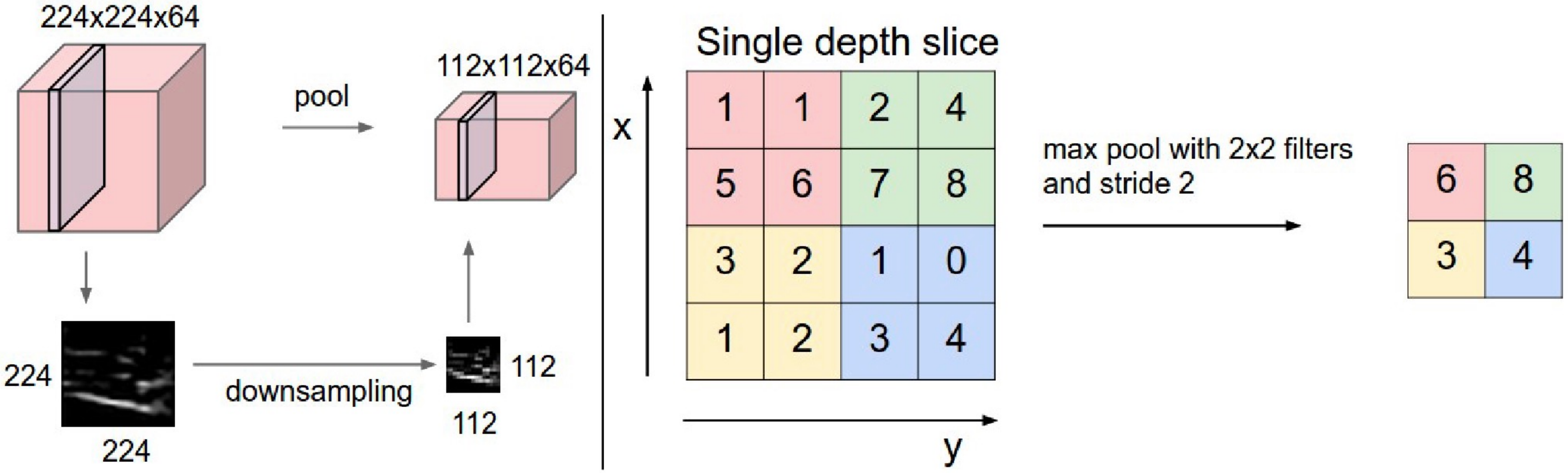
- What if we use FC to compute the same number of outputs? (the parsimony of **local connections**)
  - 230,187 * 290,400 = 66,846,304,800 parameters

- Conv layer can be seen as imposing **inductive bias** specialized for images

- This also prevents overfitting: idiosyncratic pattern that appear in few images are not picked up while training! => useless filters are 'squeezed out' or 'crowded out' by useful filters.

Pooling layer

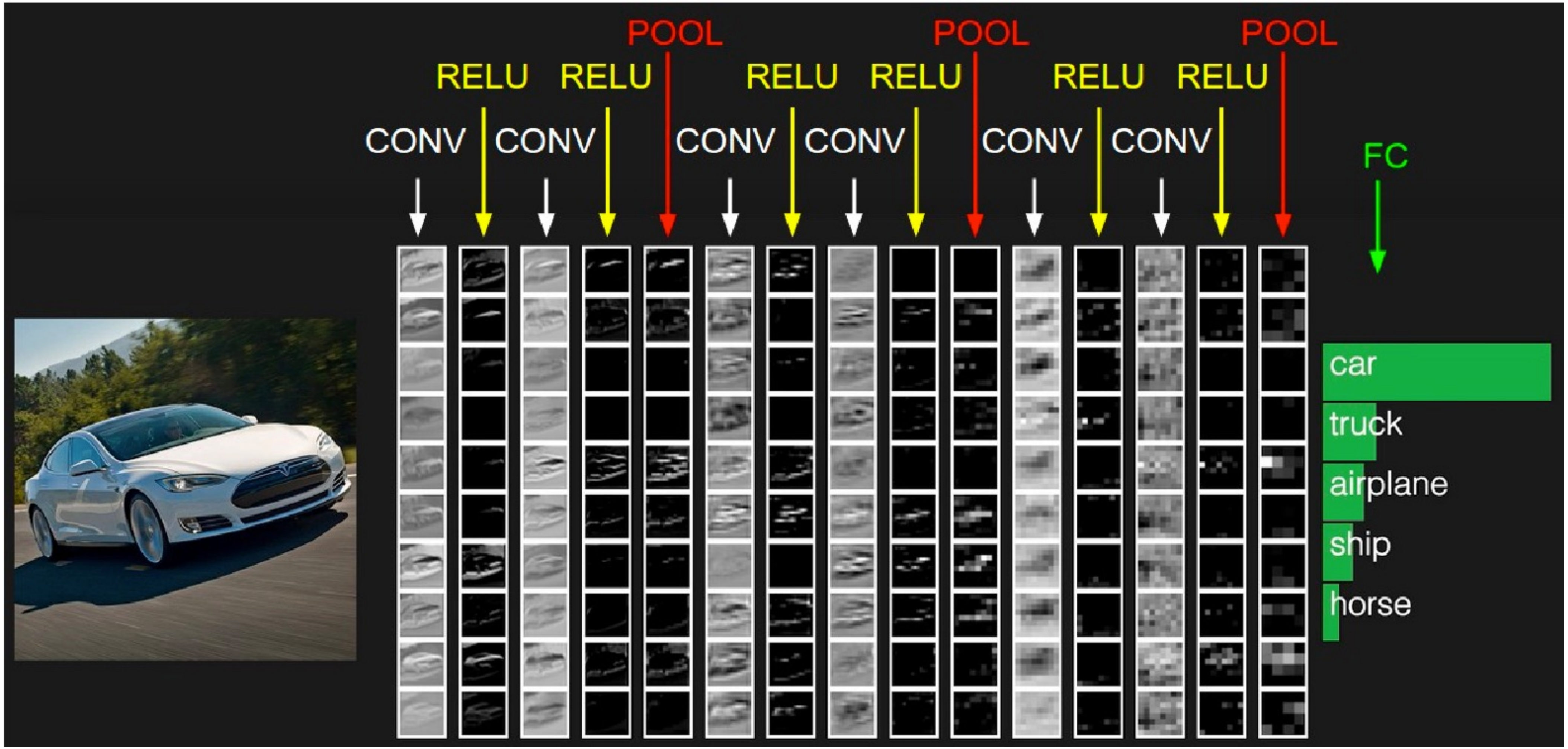
- The role: Summarize the input and scale down the spatial size.
  - has the effect of <u>**routing**</u> the region with the most activation.
- Recall <u>depth slice</u>: take the matrix at a particular depth.
- Max pooling: run a particular filter that computes maximum, for each depth slice.



- Variation: average pooling (but not popular).
- Recommended: Filter size F=2, stride length S=2. (F=3, S=2 is also commonly use – overlapping pooling).
- Note: There are **no parameters** for this layer!

figure from Stanford CS231n

Typical architectural patterns in CNN



CNN examples

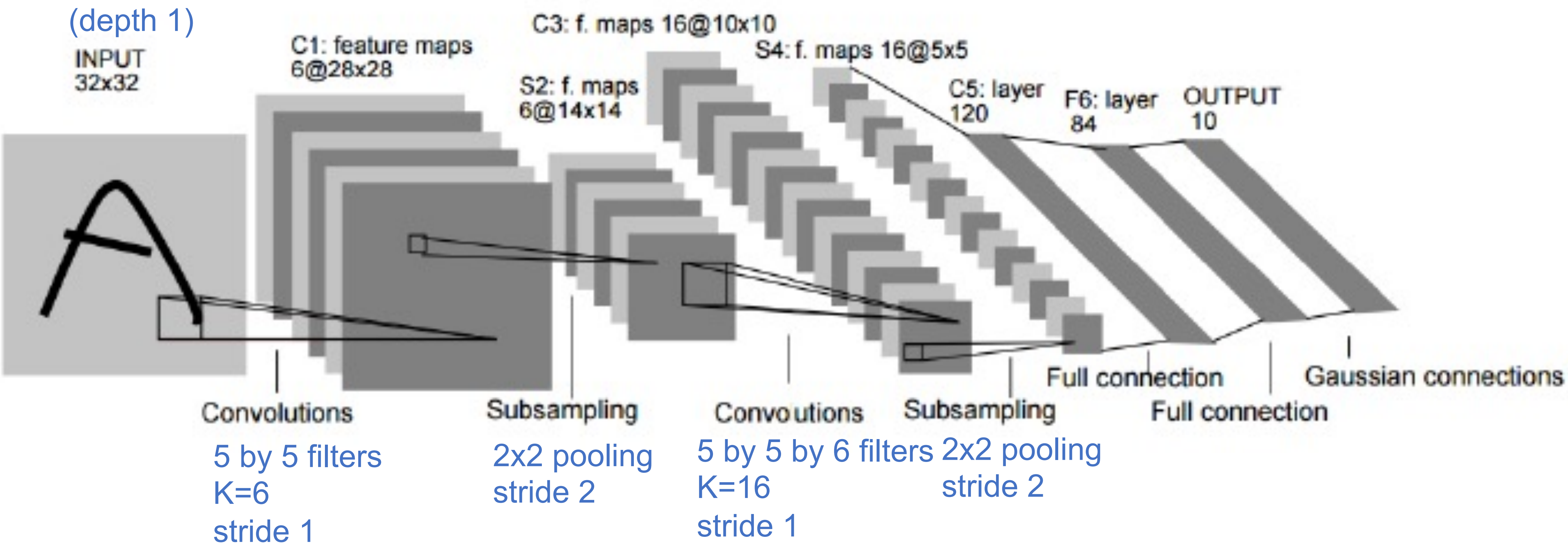
LeNet-5

- Proposed in "Gradient-based learning applied to document recognition" *, by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in Proceedings of the IEEE,* ***1998***

- Apply convolution on 2D images (MNIST) and use backpropagation

- Structure: 2 convolutional layers (with pooling) + 3 fully connected layers
  - Input size: 32x32x1
  - Convolution kernel size: 5x5
  - Pooling: 2x2

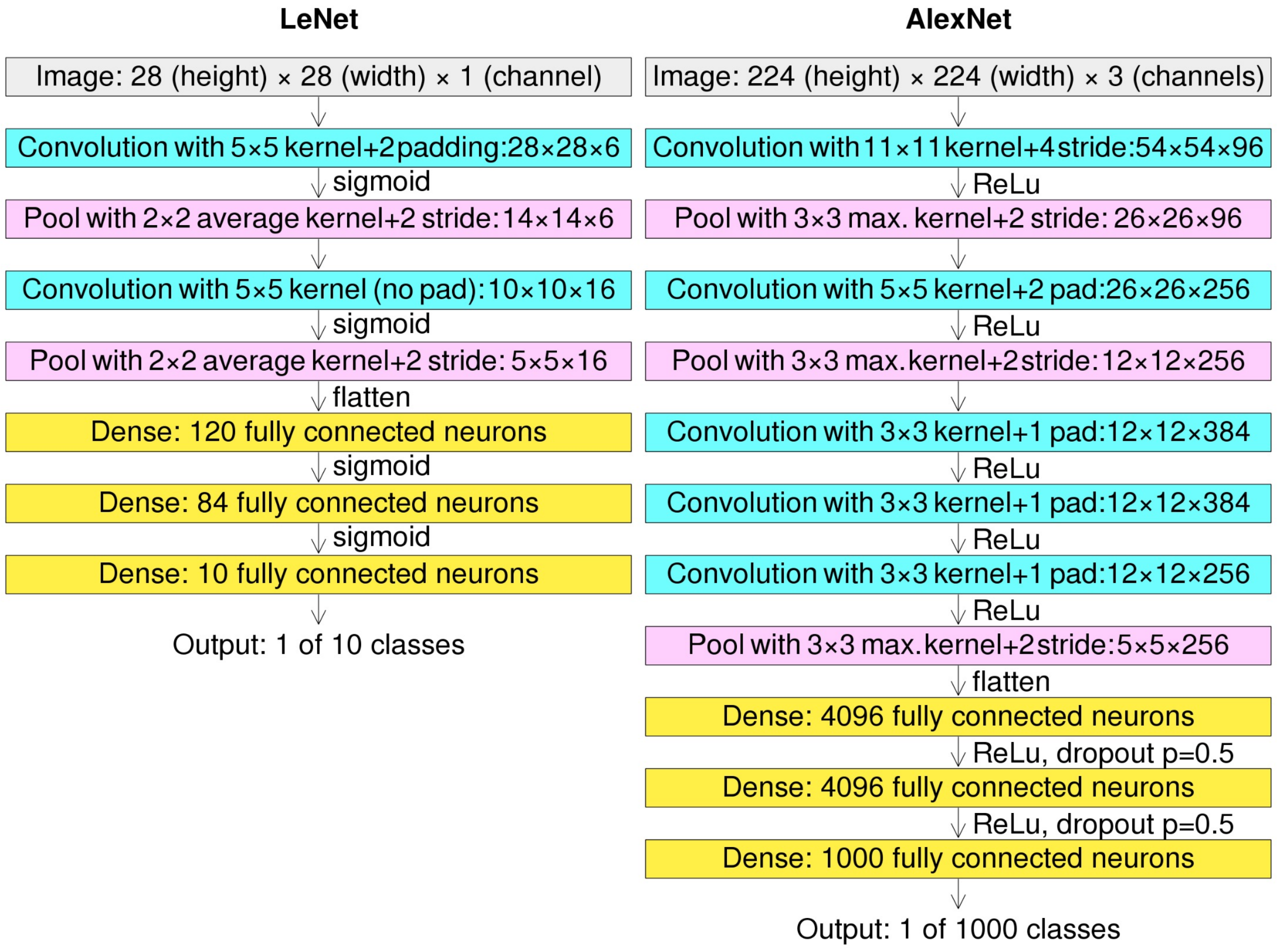
LeNet-5



"Gradient-based learning applied to document recognition" , *by Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, in Proceedings of the IEEE*

AlexNet (2012)

- Won the ImageNet competition with <u>top-5 test error rate</u> of 16.4% (second place was 26.2%). (1000 classes)
- Almost just an extension of LeNet-5. But, uses ReLU for the first time.



https://en.wikipedia.org/wiki/AlexNet

Krizhevsky, Sutskever, and Hinton, ImageNet Classification with Deep Convolutional Neural Networks, 2012.

VGGNet (2014): 7.3% error on ImageNet

- Mimic large convolutional filters with multiple small (3x3) convolutional filters
- Every time it halves the spatial size, double the # of filters

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	cc cc
maxpool			
conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	co co
maxpool			
conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	co co co **co**
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co **co**
maxpool			
conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	co co co **co**
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

[Simonyan and Zisserman, 2014]

slide from Stanford CS231n

ResNet (2016): 3.5% error on ImageNet

- Proposed in “Deep residual learning for image recognition” by *He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun*. In *Proceedings of the IEEE conference on computer vision and pattern recognition*,. 2016.

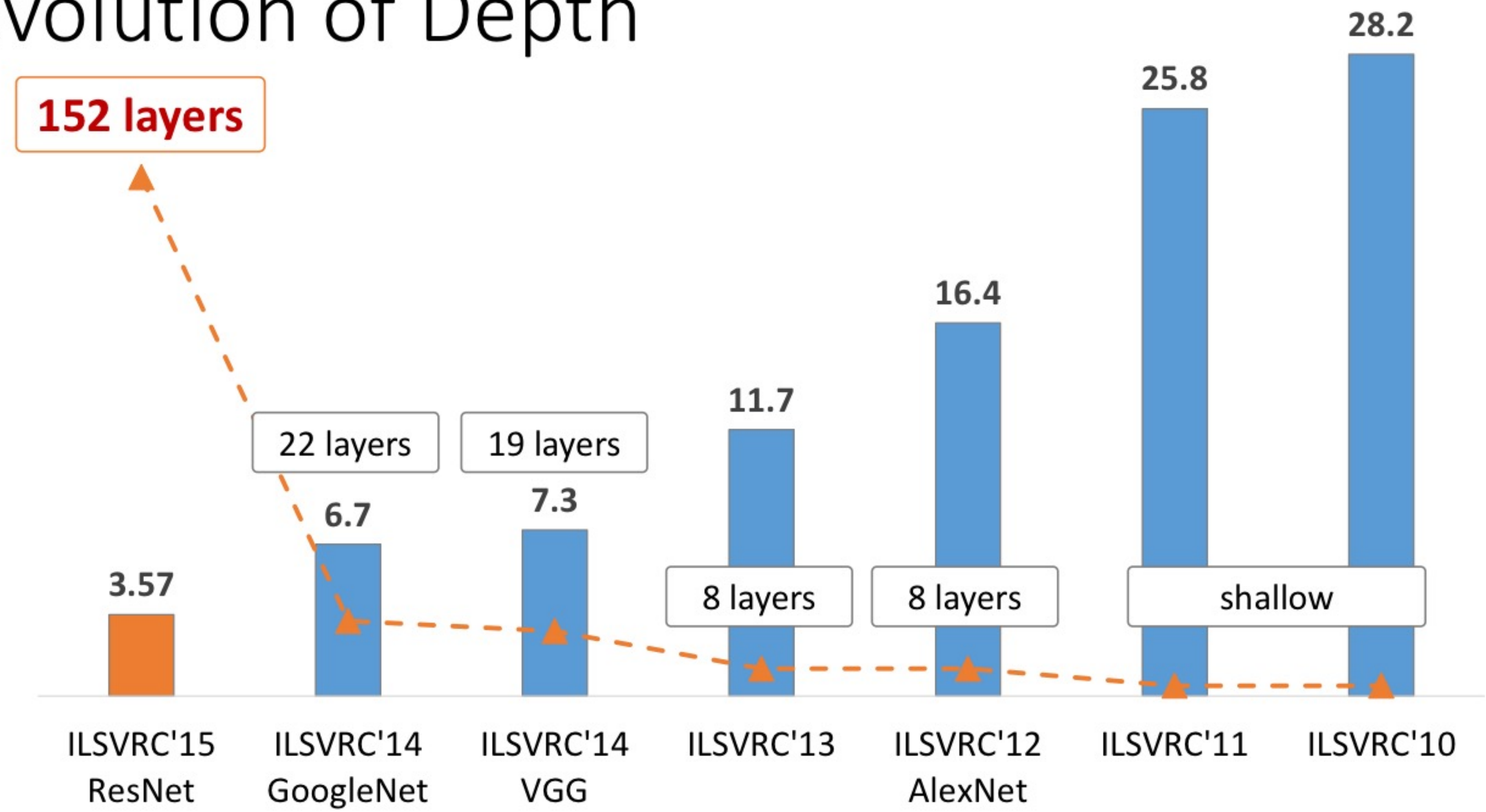
- Apply very deep networks with repeated **residual blocks**.

- Structure: simply stacking residual blocks, but the network is very deep.

- Let’s see the motivation.

Microsoft Research

Revolution of Depth

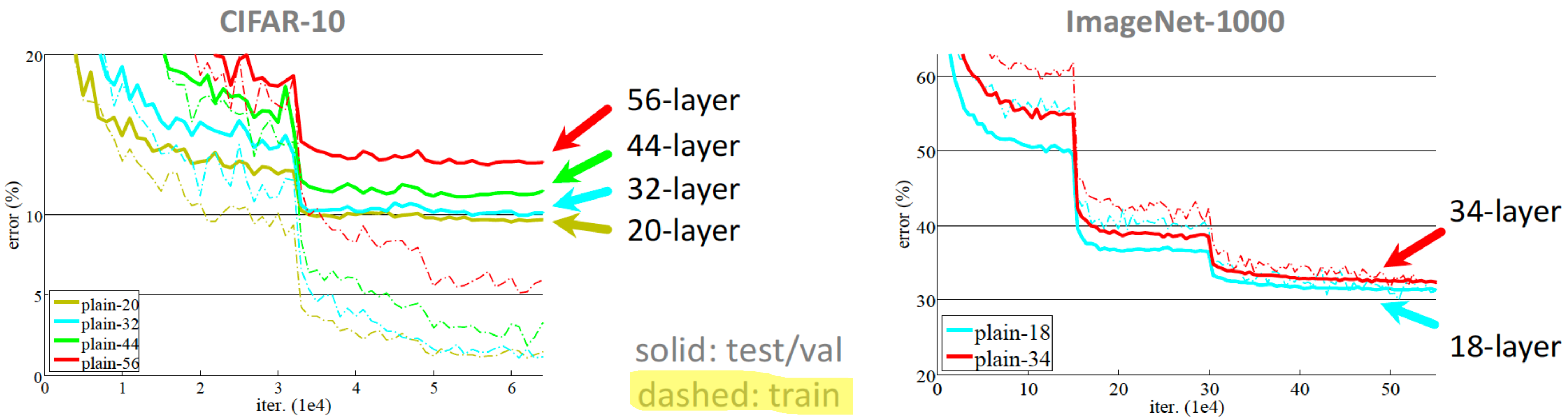


ImageNet Classification top-5 error (%)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015.

http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

Deep nets seem to suffer



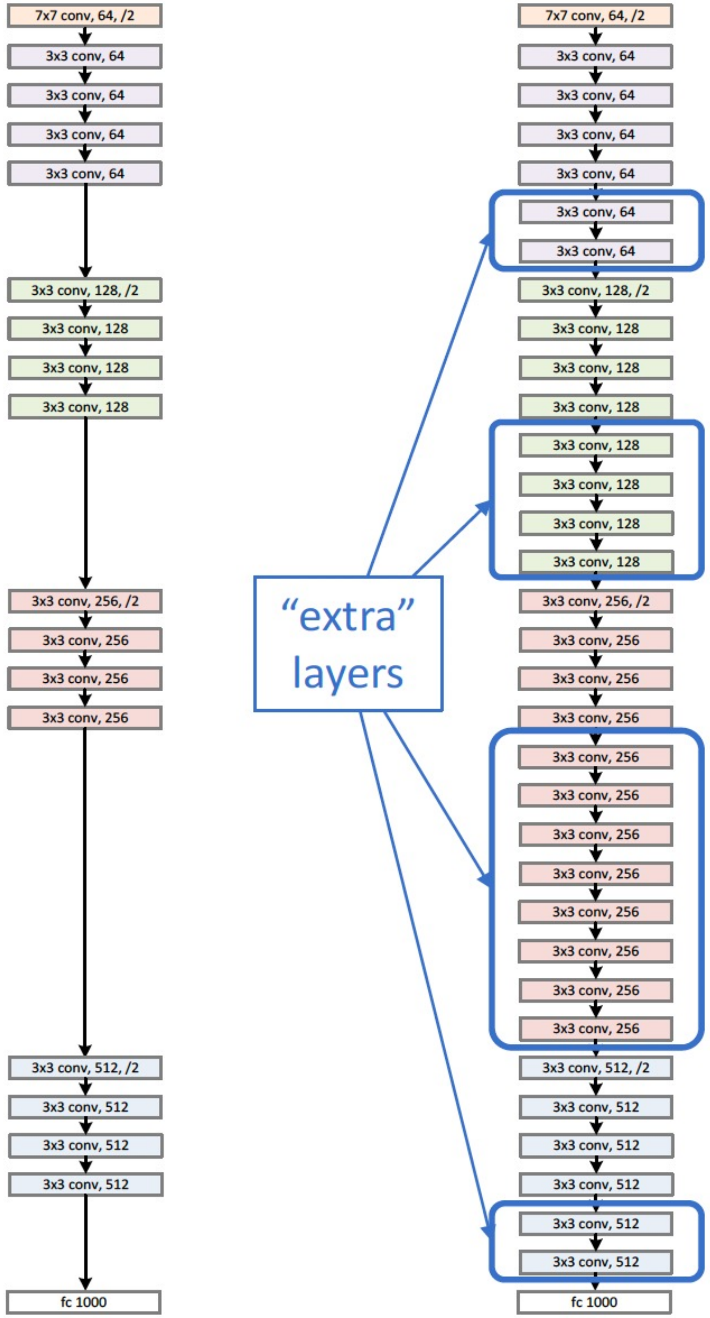
- “Overly deep” plain nets have **higher training error**
- A general phenomenon, observed in many datasets

http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

Microsoft Research

(slides from Kaiming He

a shallower model (18 layers)



a deeper counterpart (34 layers)

- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error
- Optimization difficulties: solvers cannot find the solution when going deeper…

http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

ICCV15 International Conference on Computer Vision

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

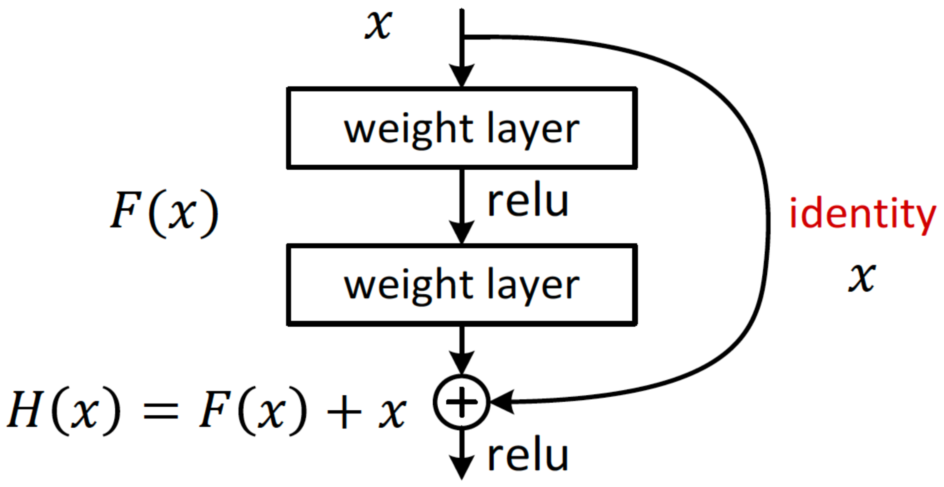
Skip connections for better optimization

- **Skip connections**

- $F(x)$ encodes residual representations, which has previously been explored in early works

- When backprop'ing, by the chain rule, gradients will 'flow' directly to the previous layer.
  - Recall: when the computation graph splits, the gradient is a summation of the gradients of the branches.
  - In contrast, plain CNNs suffer from vanishing gradient problem

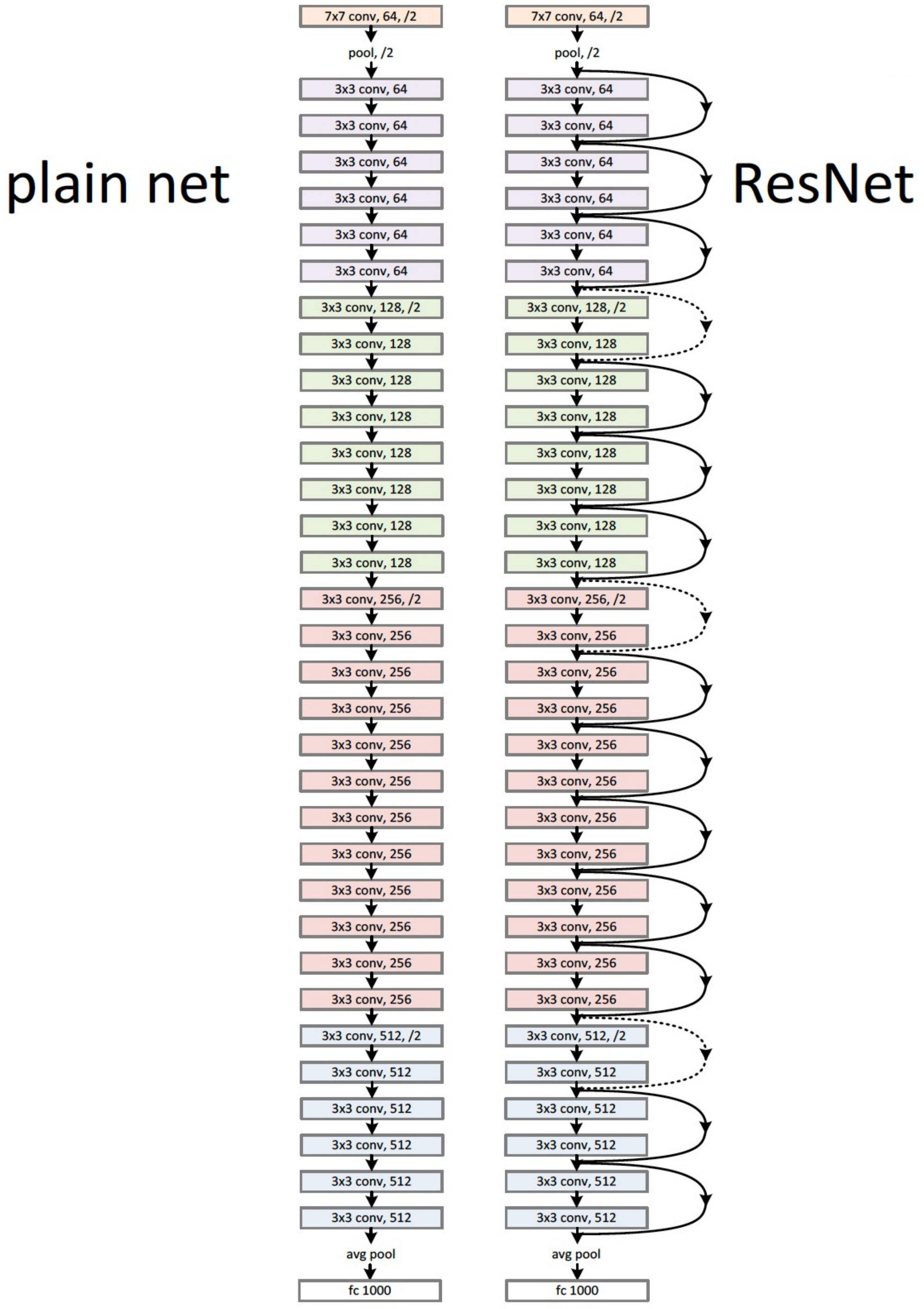
- Residual net



http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

ResNet

- VGG-style scheme: halve the special size, double the # of filters
- Max pool appears only once.
- Use conv layer with stride 2 occasionally to reduce the spatial dimension => called "**bottleneck**" blocks.

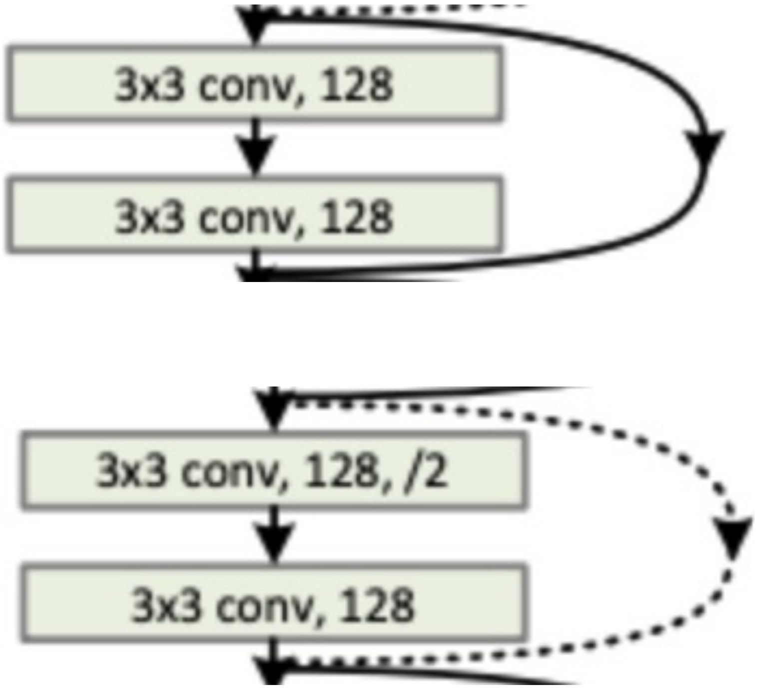


http://image-net.org/challenges/talks/ilsvrc2015_deep_residual_learning_kaiminghe.pdf

ResNet in PyTorch

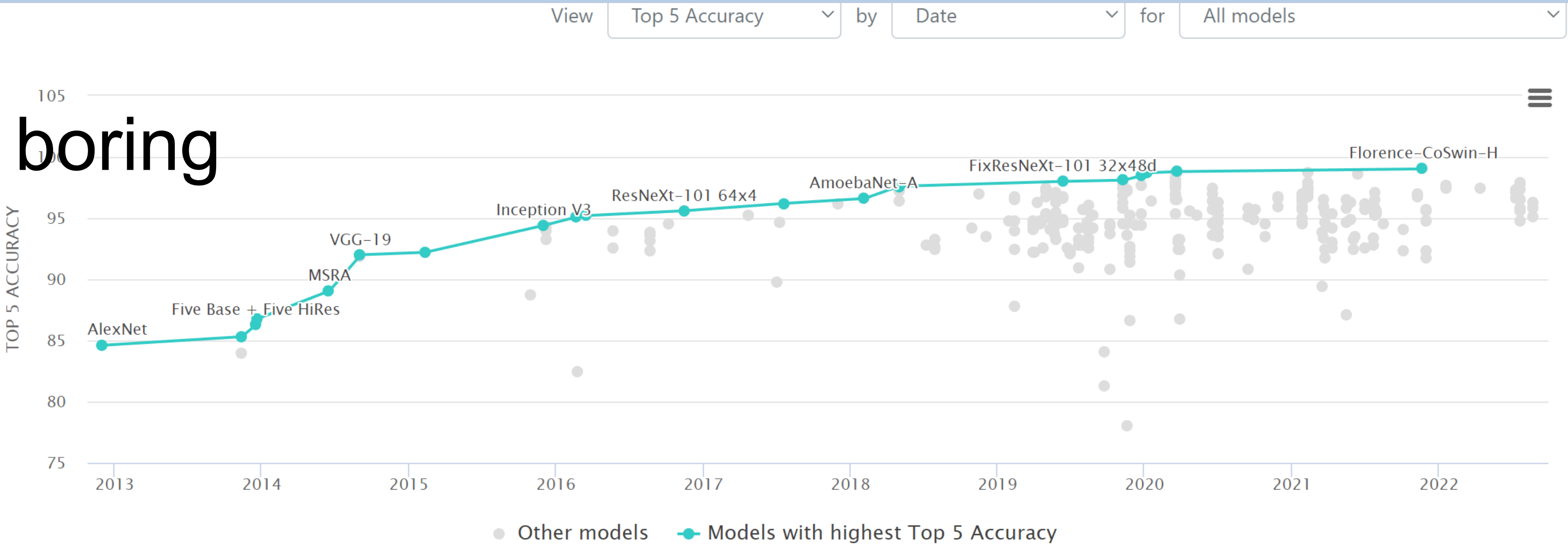
Torchvision implementation:
https://pytorch.org/vision/0.8/_modules/torchvision/models/resnet.html

```python
class Bottleneck(nn.Module):
        def forward(self, x):
            identity = x

            out = self.conv1(x)
            out = self.bn1(out)
            out = self.relu(out)

            out = self.conv2(out)
            out = self.bn2(out)
            out = self.relu(out)

            out = self.conv3(out)
            out = self.bn3(out)

            if self.downsample is not None:
                identity = self.downsample(x)

            out += identity
            out = self.relu(out)

            return out
```

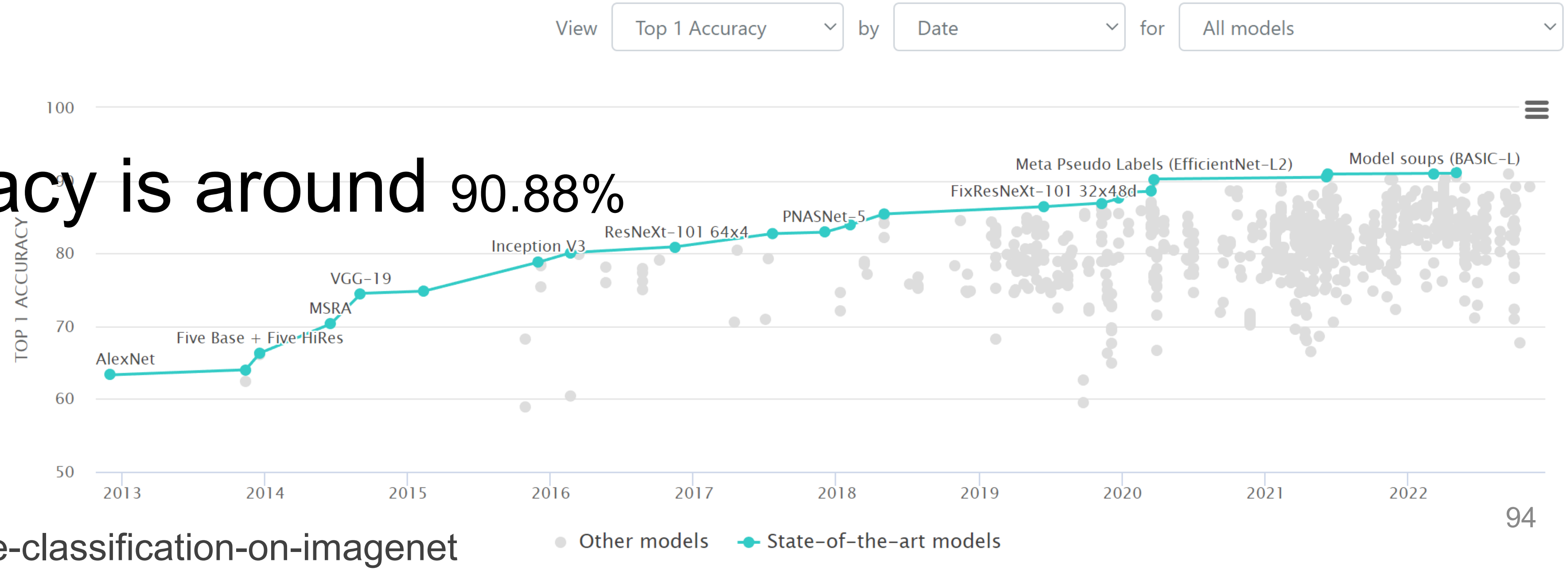


ImageNet nowadays

Top-5 accuracy is boring



SoTA top-1 accuracy is around 90.88%



https://paperswithcode.com/sota/image-classification-on-imagenet

Autoencoder

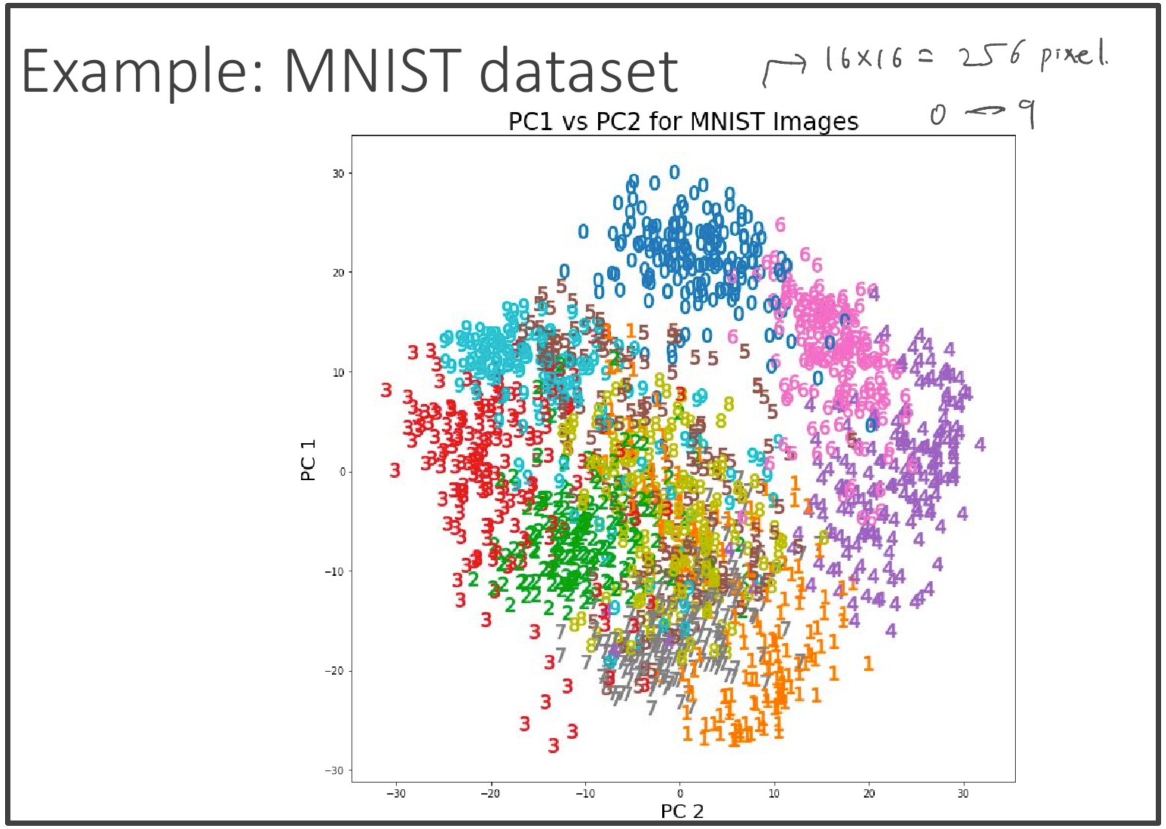
Unsupervised Learning Review

- Recall: unlabeled data.
- Q: what is the main goal of unsupervised learning?
- Examples: clustering, PCA.

- Recall PCA can be used for 'representation learning' = learning useful (and compact) features.

  (learned features = projected feature vector)

- NNs can be used to do generalizations of PCA.



Introductory Example

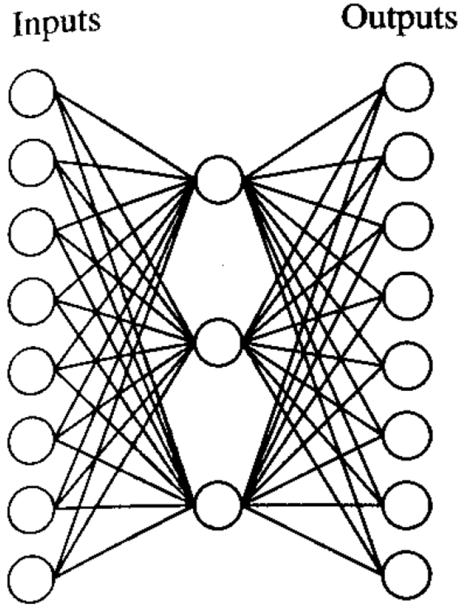
- Suppose you have a number in {0,1,2,3,4,5,6,7}
- What would be a compact representation (say, for computers)?

- Q: how many bits do we need?

Early Observations

Train a neural net by imposing squared loss on all the output units & backpropagation.

Q: What do the hidden values look like?

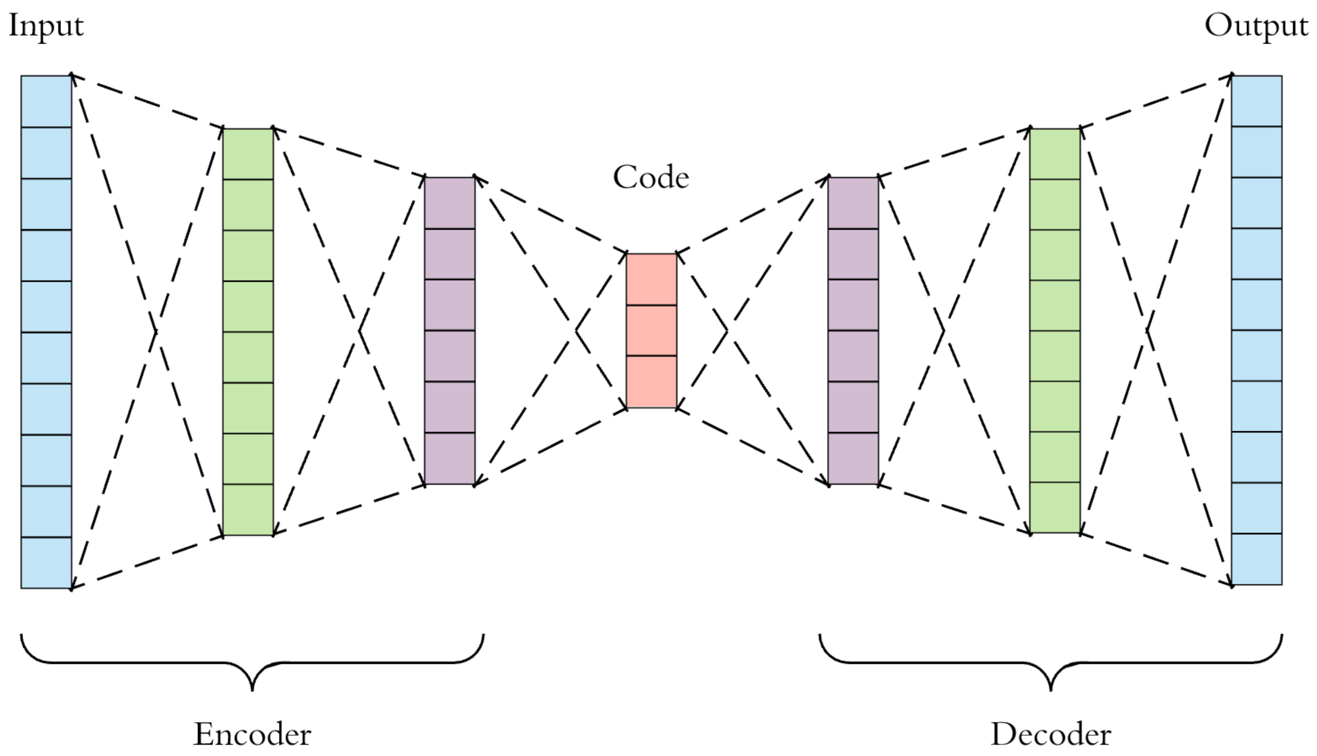


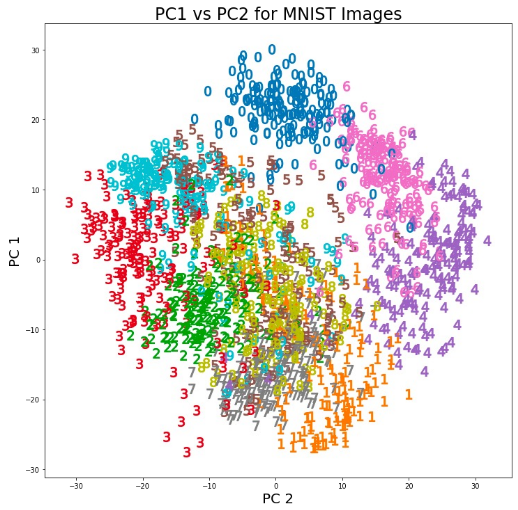
Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

p107, Tom Mitchell, “Machine Learning”

Autoencoder using deep networks





We can do this for any data!

How to use it:
- Encoder: for dimensionality reduction
- Decoder: generate new samples from the distribution by varying the input 'code'

image from https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

PCA as a linear neural network

linear = no activation

PCA pseudocode

- Input: data matrix $X \in \mathbb{R}^{n \times d}$
- Preprocess: Let $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$. Compute $x_i' = x_i - \mu, \forall i \in [n]$
- Compute the top k eigenvectors $V = [v_1, \dots, v_k]$ of $\frac{1}{n}\sum_{i=1}^{n} x_i' (x_i')^\top$
- Feature map: $\phi(x) = \boxed{\left(v_1^\top (x-\mu), \dots, v_k^\top (x-\mu)\right)} \in \mathbb{R}^k$ coefficient.
- Decorrelating property: ("whitening").
  - $\frac{1}{n}\sum_{i=1}^{n} \phi(x_i) = 0$
  - $\frac{1}{n}\sum_{i=1}^{n} \phi(x_i)\phi(x_i)^\top = Diag(\lambda_1, \dots, \lambda_k)$

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ \vdots & & \ddots & 0 \\ & & & \lambda_k \end{bmatrix}$$

- Reconstruction (the actual projection): apply $\mu + V\phi(x)$

$(v_1 \cdots v_k)$

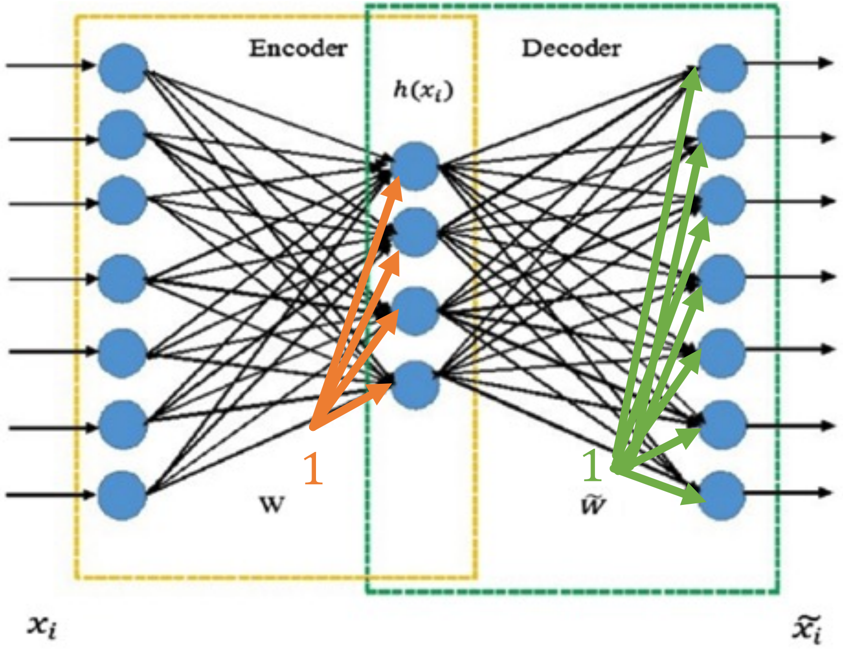
PCA as a <u>linear</u> NN

- Feature map: $\phi(x) = \left(v_1^\top(x-\mu), \dots, v_k^\top(x-\mu)\right) \in \mathbb{R}^k$ coef
- Decorrelating property: ("whitening")
  - $\frac{1}{n}\sum_{i=1}^{n} \phi(x_i) = 0$
  - $\frac{1}{n}\sum_{i=1}^{n} \phi(x_i)\phi(x_i)^\top = Diag(\lambda_1, \dots, \lambda_k)$
- Reconstruction (the actual projection): apply $\mu + V\phi(x)$

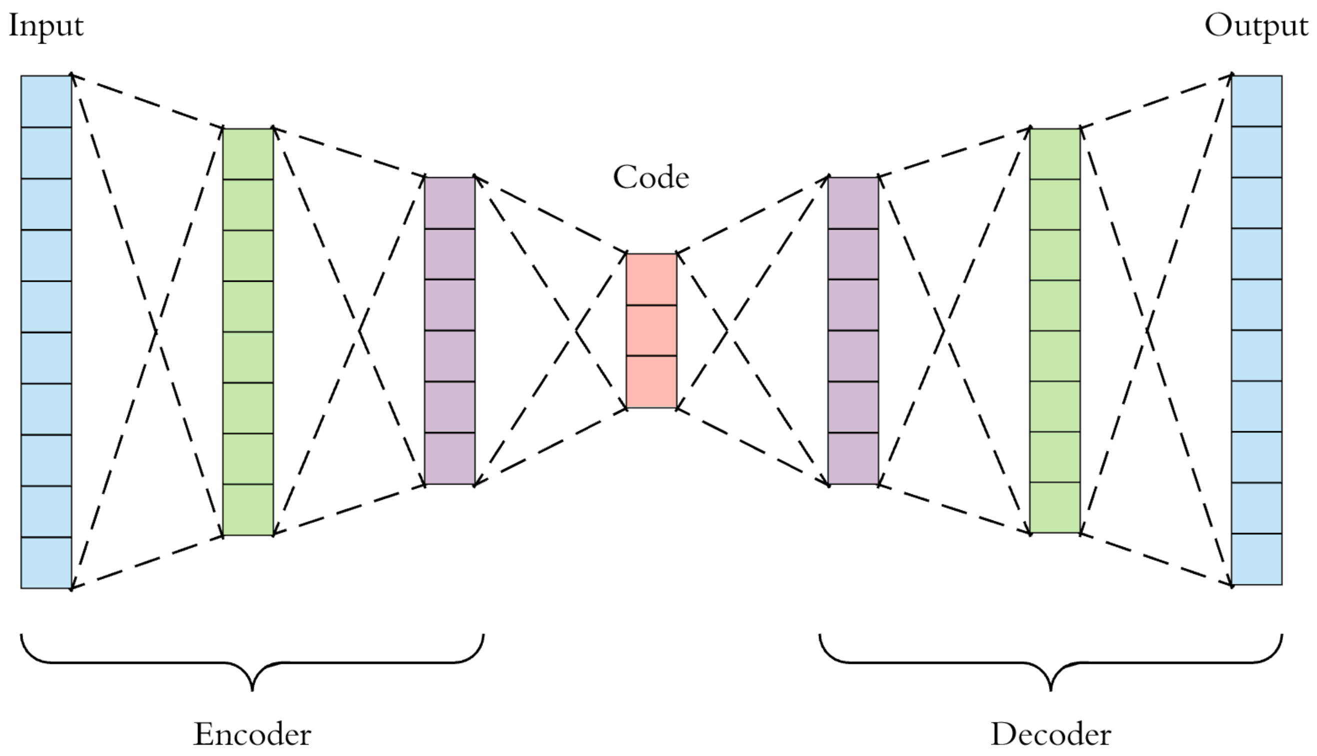
- k units in the hidden layer.
- The PCA can be represented as a NN (with constant bias added in each layer):

  - Encoder: $h = \begin{pmatrix} - & v_1 & - \\ & \cdots & \\ - & v_k & - \end{pmatrix} \cdot x + \begin{pmatrix} -v_1^\top \mu \\ \cdots \\ -v_k^\top \mu \end{pmatrix}$

  - Decoder: $\tilde{x} = \begin{pmatrix} | & & | \\ v_1 & \cdots & v_k \\ | & & | \end{pmatrix} \cdot h + \begin{pmatrix} | \\ \mu \\ | \end{pmatrix}$



Autoencoder using deep networks



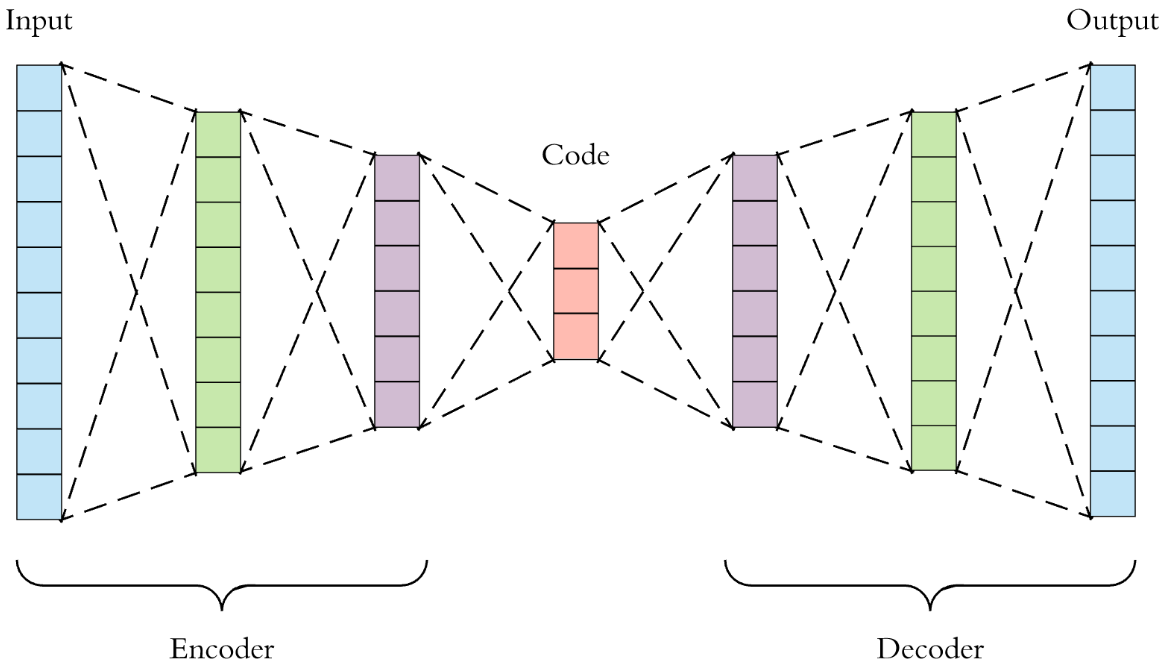
We can do this for any data!

<u>What about images?</u>

image from https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Training autoencoders

- Given:
  - data $x_1, \ldots, x_n \in \mathbb{R}^d$,
  - Embedding dimension k ($k \ll d$)



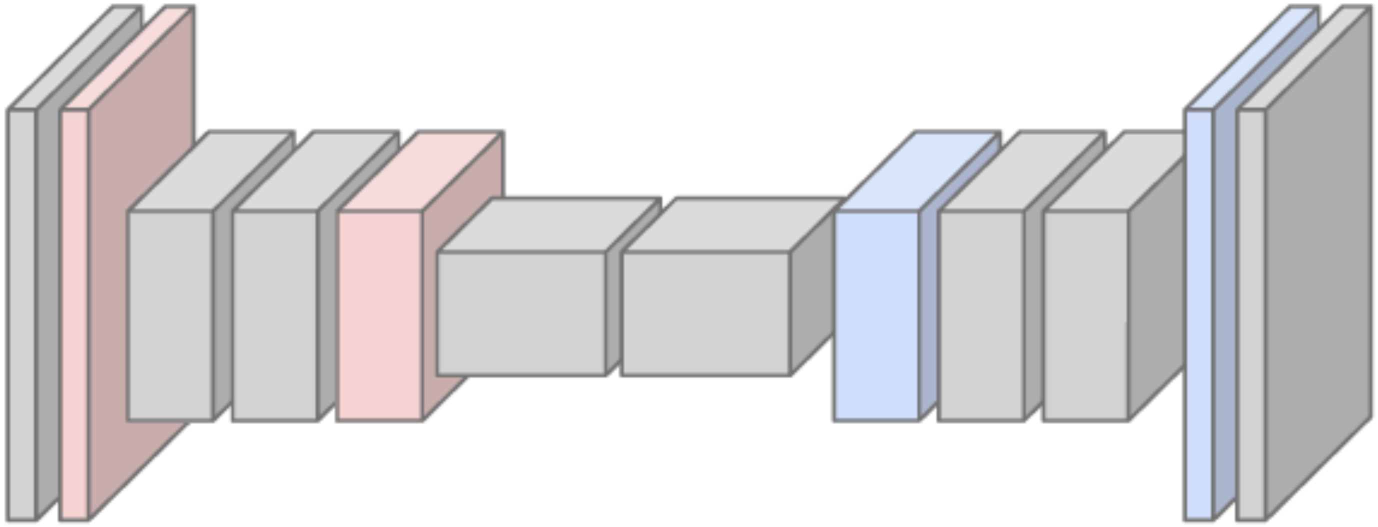
- Goal: obtain
  - Encoder network $f_\theta: \mathbb{R}^d \to \mathbb{R}^k$
  - Decoder network $g_\phi: \mathbb{R}^k \to \mathbb{R}^d$
  - Such that for every i, $x_i \approx g_\phi(f_\theta(x_i))$

- Most commonly used formulation (can be straightforwardly trained by gradient-based methods):

$$\mathrm{minimize}_{\theta,\phi} \sum_{i=1}^{n} \underbrace{\left\| x_i - g_\phi\left(f_\theta(x_i)\right) \right\|}_{\text{Reconstruction error}}{}^2$$

Autoencoder for images

- Encoder: conv-conv-pool-conv-conv-pool-…,
- Decoder: conv-conv-pool-…?? It will reduce the spatial dimension rather than increasing it.
- How to do the opposite of pooling (or conv with stride length >= 2)?



Following slides largely based on Stanford cs231n https://youtu.be/nDPWywWRIRo?t=1109

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf

“Un”pooling

Nearest Neighbor

Input: 2 x 2

1	2
3	4

Output: 4 x 4

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

“Bed of Nails”

Input: 2 x 2

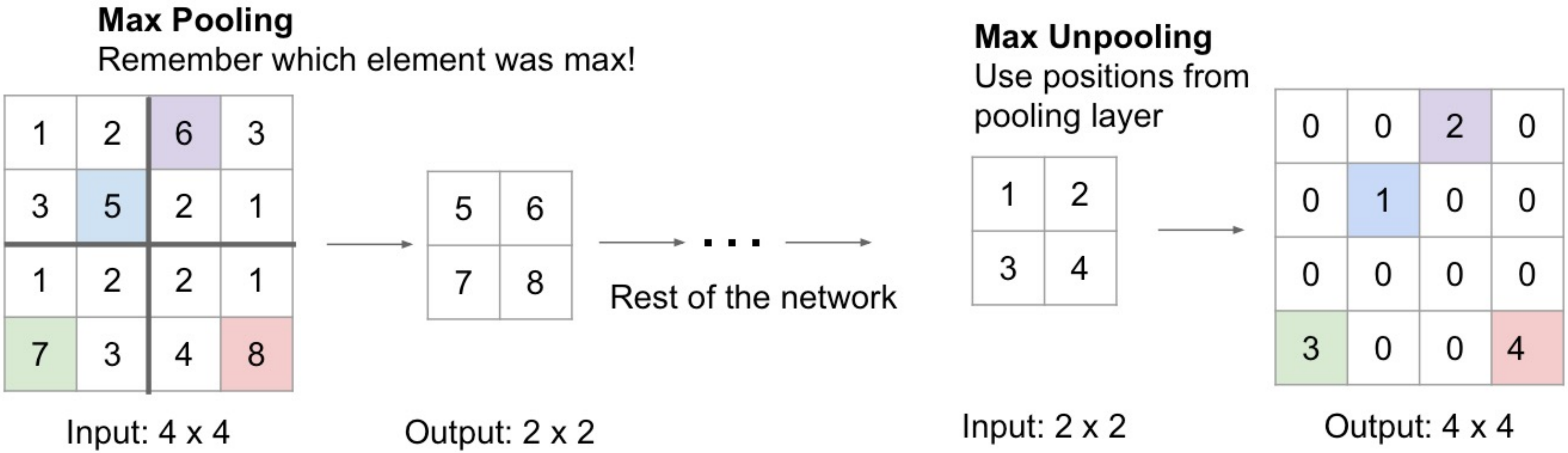
1	2
3	4

Output: 4 x 4

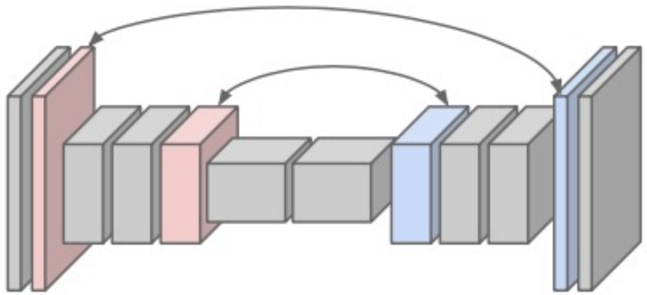
1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

(fig. from Stanford cs231n)

Max unpooling



Corresponding pairs of downsampling and upsampling layers

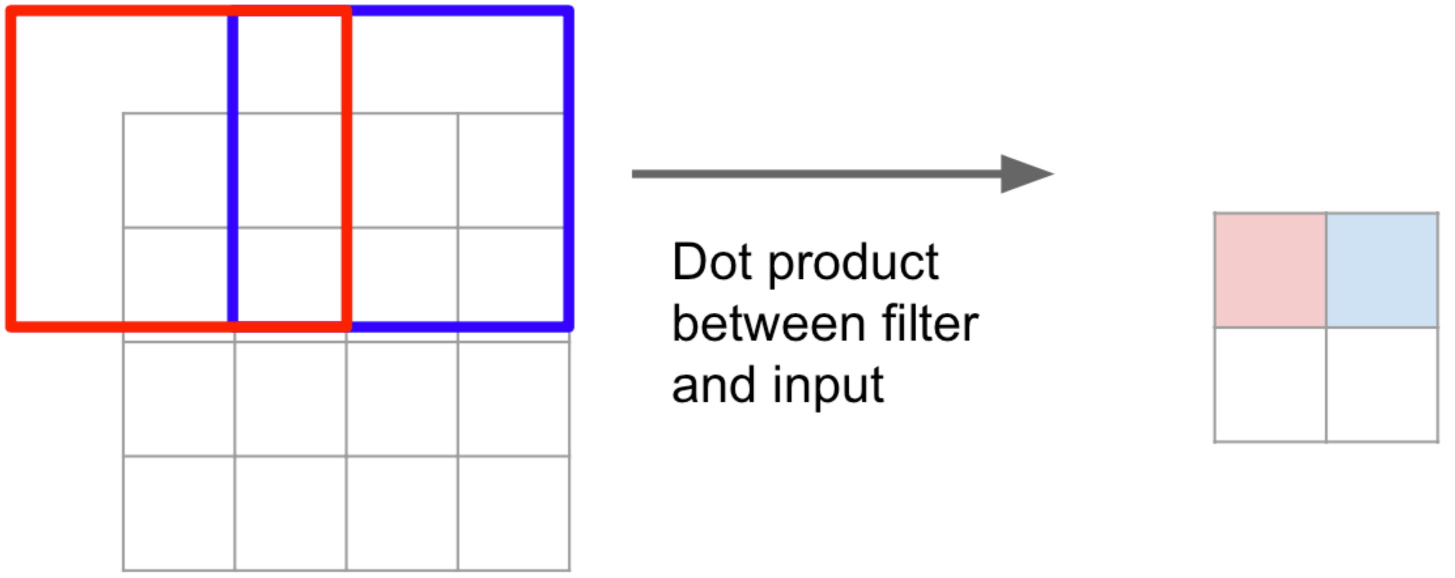


The network must be symmetric

(fig from Stanford cs231n)

Transposed convolution

- Other names: upconvolution, fractionally strided convolution, backward strided convolution, deconvolution (don't use this name)
- Recall: 3 x 3 convolution with stride 2 pad 1.



Input: 4 x 4

Output: 2 x 2

Filter moves 2 pixels in the input for every one pixel in the output

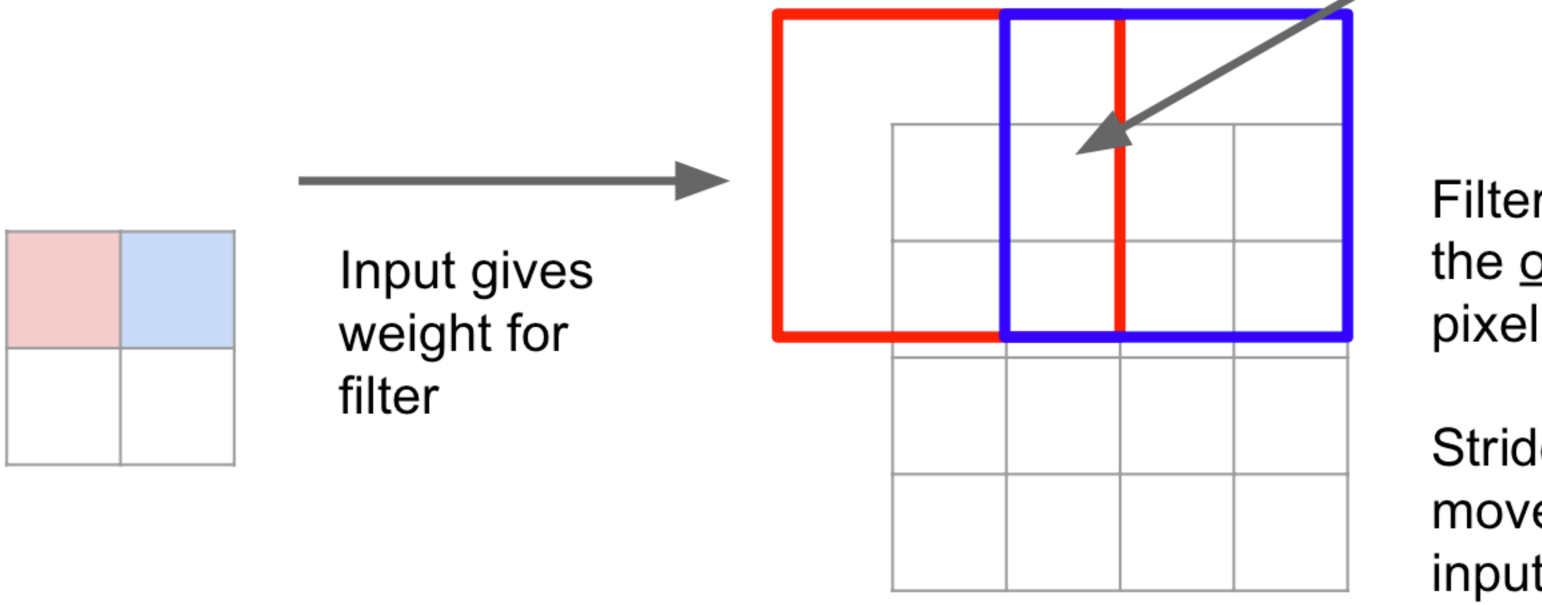
Stride gives ratio between movement in input and output

(fig from Stanford cs231n)

Transposed convolution

3 x 3 **transpose** convolution, stride 2 pad 1

Sum where output overlaps



Input: 2 x 2

Output: 4 x 4

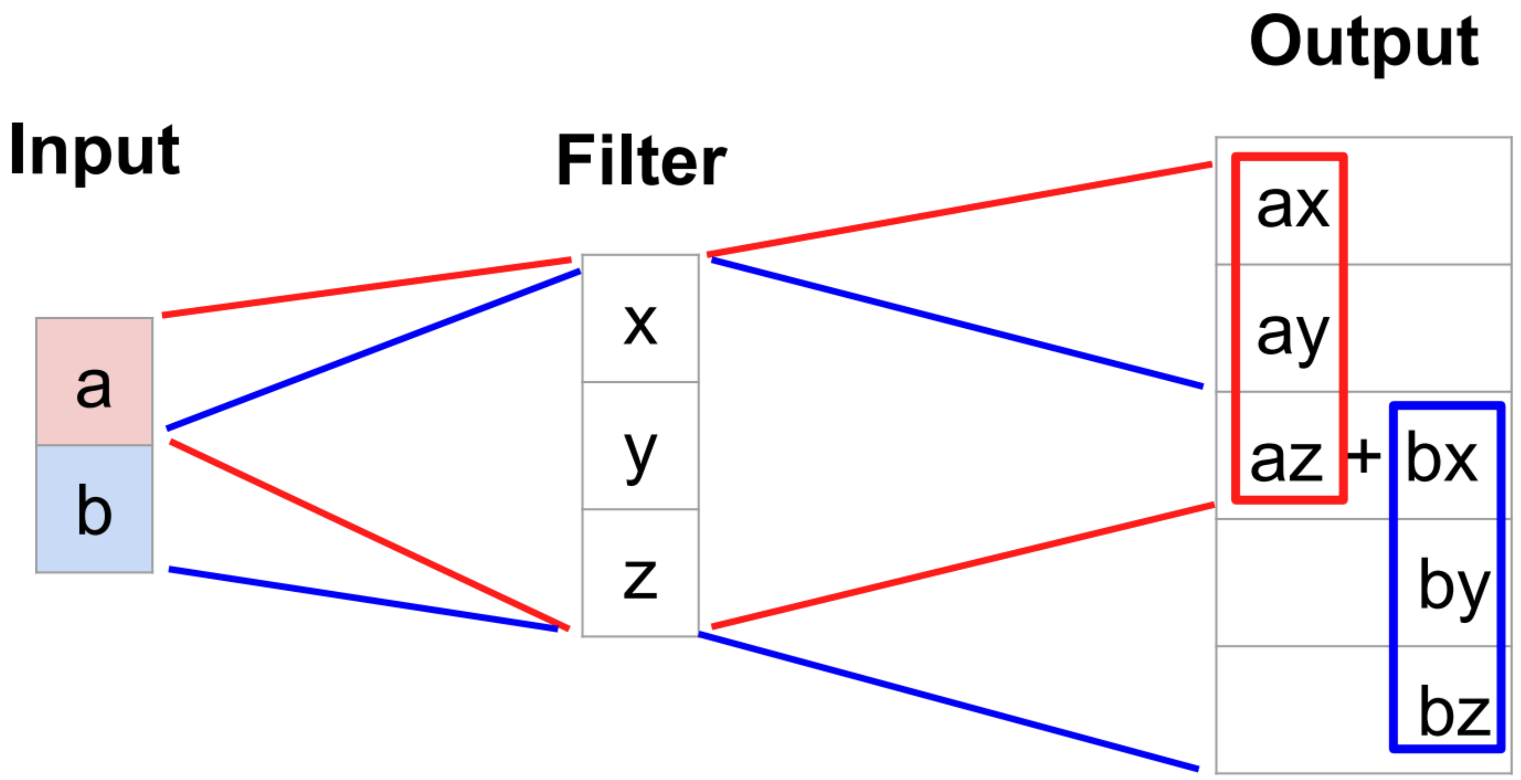
Filter moves 2 pixels in the <u>output</u> for every one pixel in the <u>input</u>

Stride gives ratio between movement in output and input

Disclaimer: this is not the inverse of convolution!
Rather, it's just a <u>variation</u> of the convolution.

(fig from Stanford cs231n)

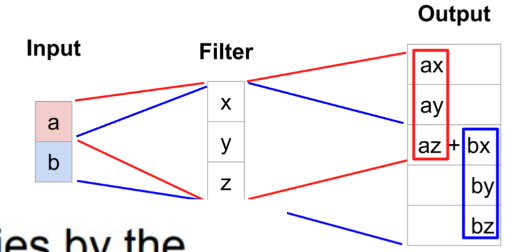
1D transposed convolution



Output contains copies of the filter weighted by the input, summing at where at overlaps in the output

(fig from Stanford cs231n)

1D transposed convolution: matrix form



We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Transposed convolution multiplies by the transpose of the same matrix:

$$\vec{x} *^T \vec{a} = X^T\vec{a}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

Example: 1D transposed conv, kernel size=3, stride=2, padding=0

(fig from Stanford cs231n)

Resources

"The Deep Learning Book" by Goodfellow et al.

https://www.deeplearningbook.org/

3Blue1Brown Youtube channel has a nice four-part intro:

https://www.youtube.com/watch?v=aircAruvnKk

Free book by Michael Nielson uses MNIST example in Python:

http://neuralnetworksanddeeplearning.com/

Prof. Stephen Bethard often teaches an excellent class:

ISTA 457 / INFO 557