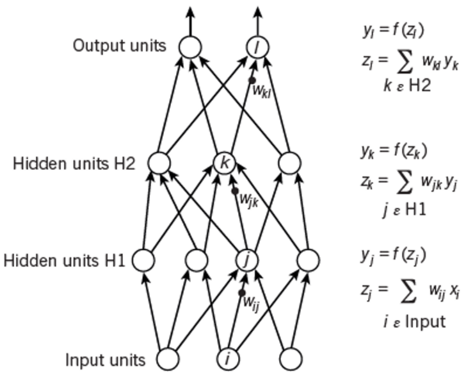
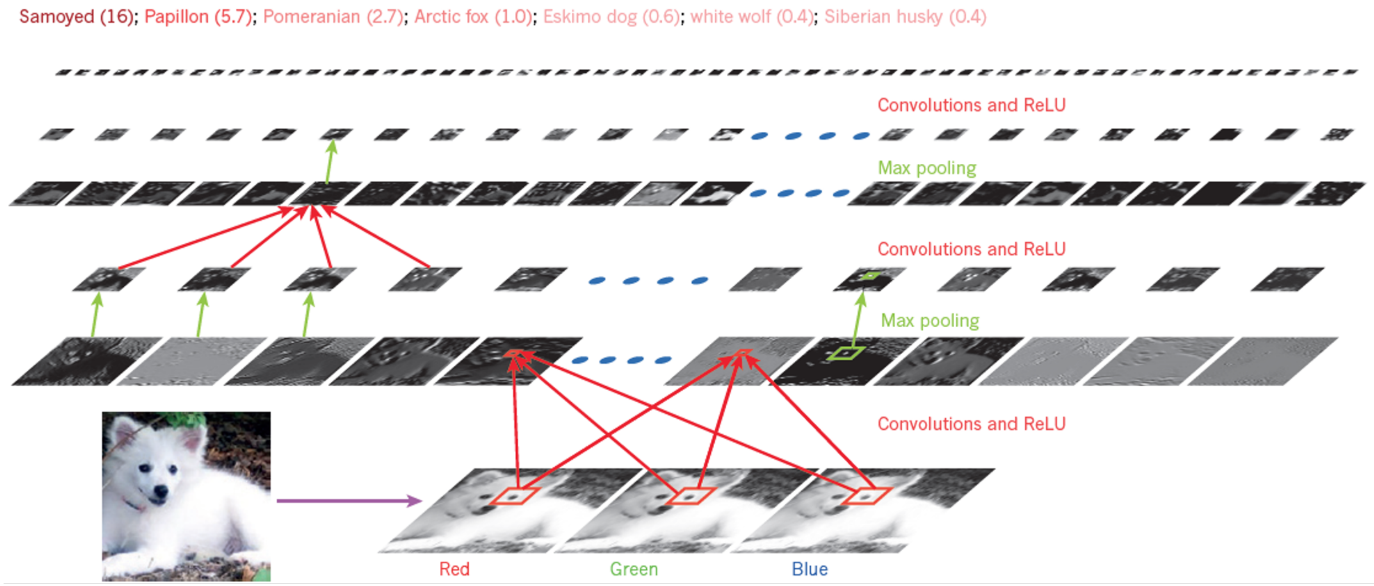
Biomedical Data Science: Mining and Modeling

Deep Learning II: Deep Supervised Learning, Feed-forward Neural Networks, Convolutional Neural Networks, and Recurrent Neural Networks

Dr. Martin Renqiang Min
NEC Laboratories America

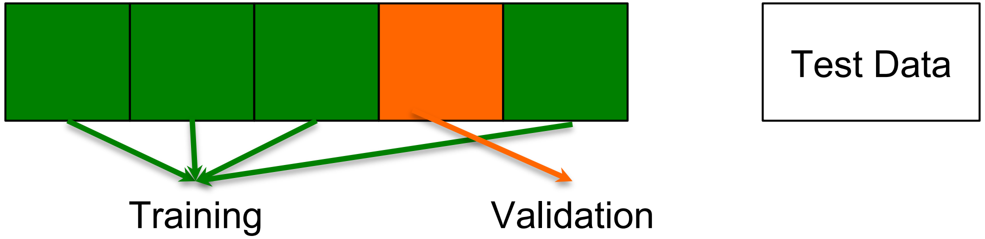
Supervised Deep Learning





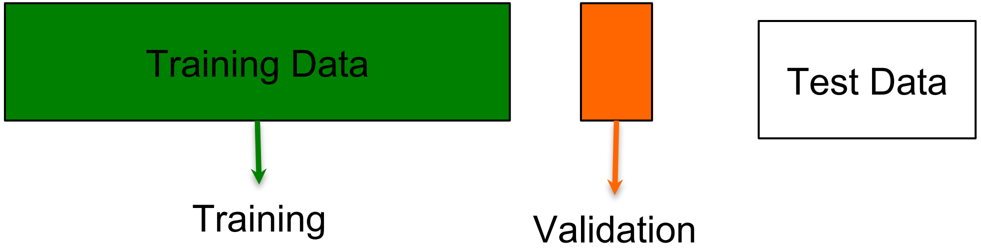
LeCun, Bengio, and Hinton, Deep Learning. Nature 2015

Supervised Deep Learning



Supervised Machine Learning:

Feature Representation +
Classification/Regression Loss +
Optimization (on training data)
→ Prediction (on test data)
(hyper-parameter tuning with n-fold CV, n=5)

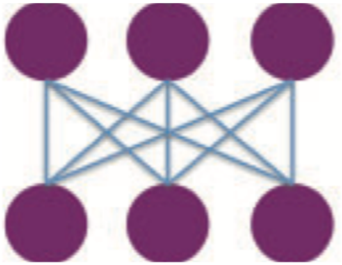


Supervised Deep Learning:

Input features and adaptively learned features by hidden layers + Mean Squared Error/Hinge Loss/Cross-Entropy Loss + SGD with Momentum (on large-scale training data)
→ Good Prediction Performance (on test data)
(hyper-parameter tuning on a validation set)

Fully Connected Layer

y

W 

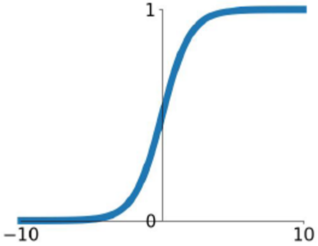
x

$$y = W x$$

Activation Functions

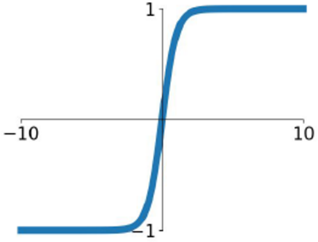
Sigmoid

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



tanh

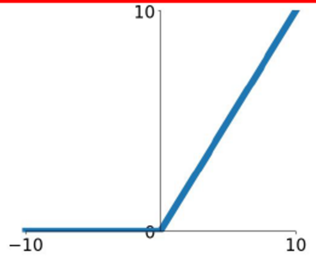
$\tanh(x)$



ReLU

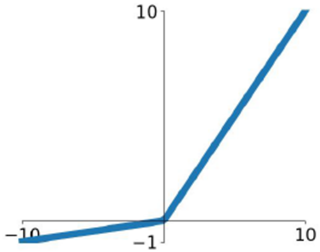
$\max(0, x)$

Good default choice



Leaky ReLU

$\max(0.1x, x)$

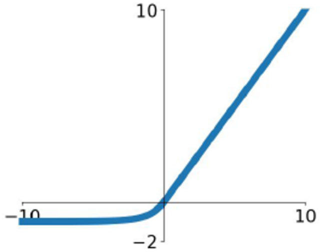


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

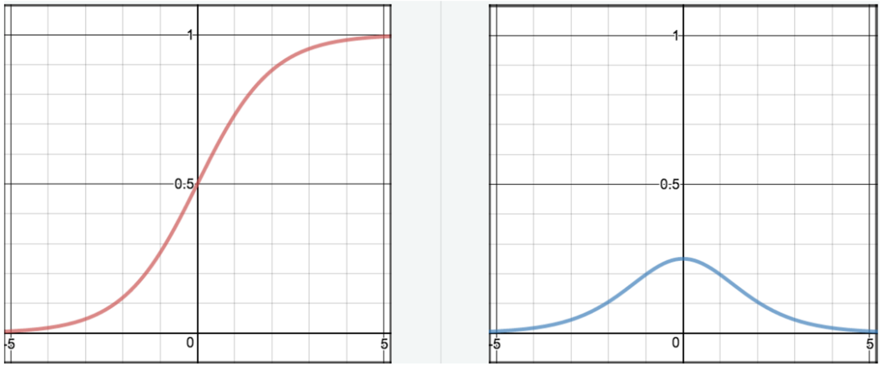
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



DNN with sigmoid and tanh activation functions has serious vanishing gradient and saturation issue

y

$$y = \frac{1}{1 + e^{-z}}$$

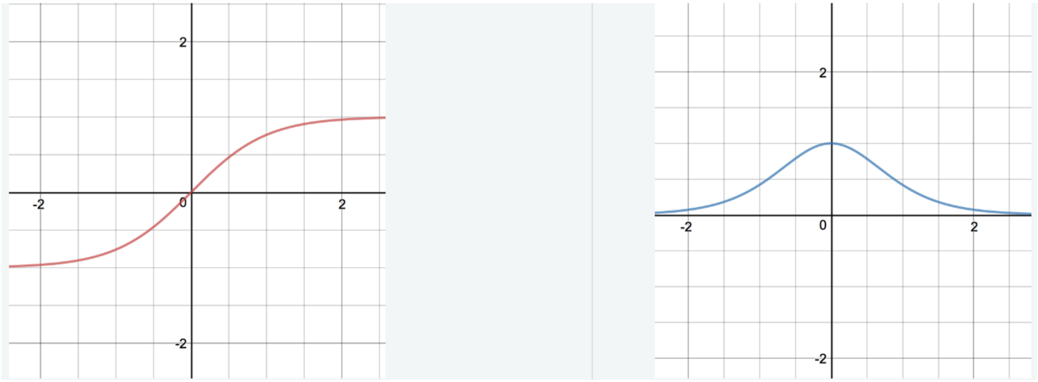


Grad = y(1-y)

z

y

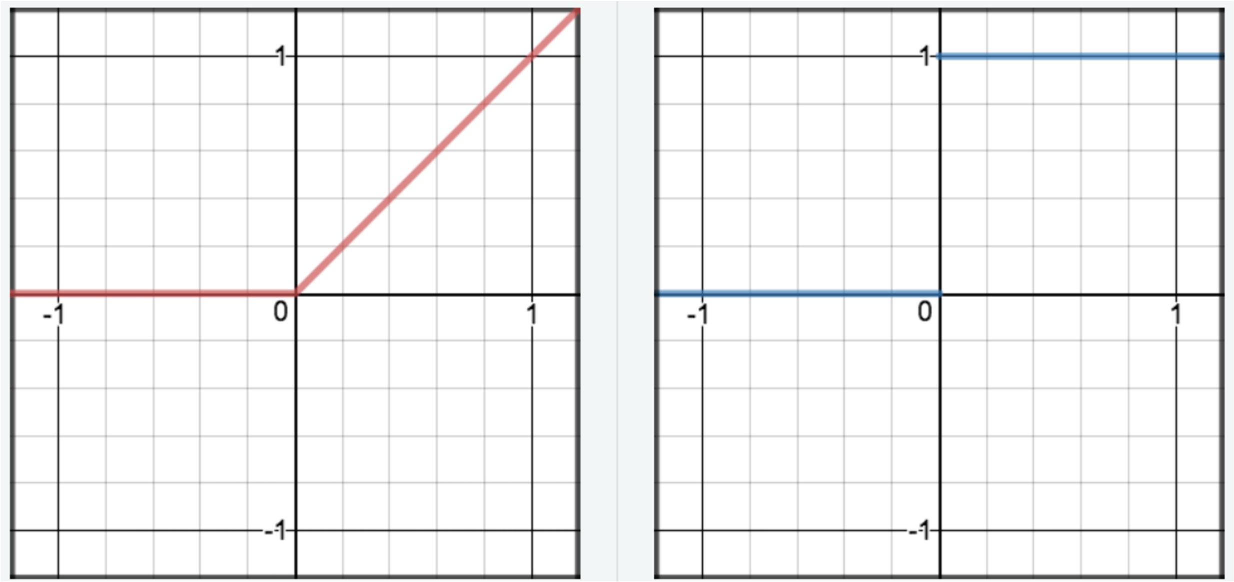
$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



Grad = 1-y^2

z

ReLU Activation Function

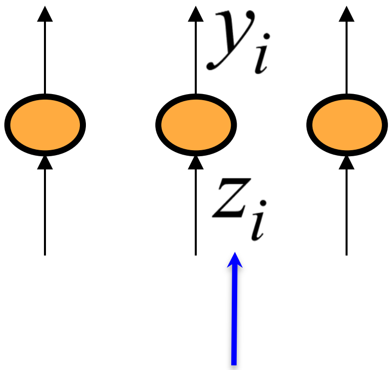


Avoid vanishing gradient and less computationally expensive than sigmoid and tanh

But it might cause dead neuron and the activity is not bounded above

Softmax Activation Function

The output units in a softmax group use a non-local non-linearity:



softmax group

this is called the “logit”

$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i \left(1 - y_i\right)$$

Often used on top of a fully connected layer, which transforms an activity vector **z** into probabilities of classifying **x** into K classes

Loss Function: Cross-Entropy Loss

The right cost function is the negative log probability of the target class.

$$C = -\sum_j t_j \log y_j$$

Target Class

C has a very big gradient when the target value is 1 and the output is almost zero.

A value of 0.001 is much better than 0.0000001

The steepness of dC/dy exactly balances the flatness of dy/dz

$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i$$

Loss Function: Mean Squared Error

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

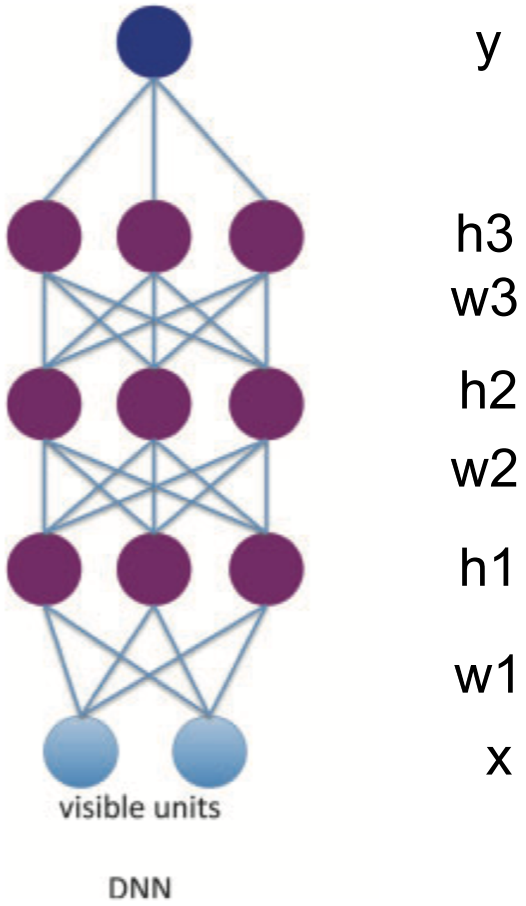
MSE is a very bad cost function for softmax output units.
Why?

Loss Function: Hinge Loss

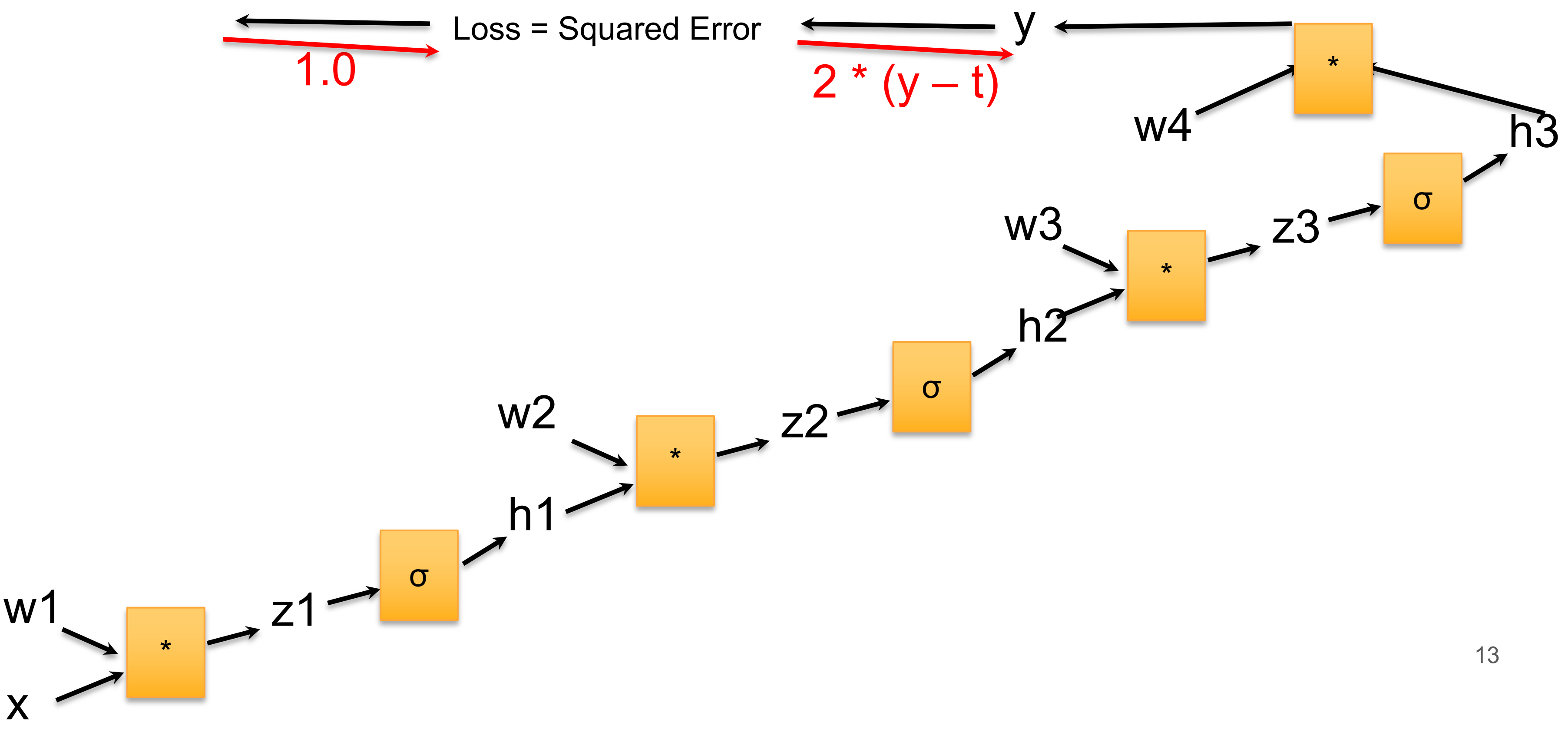
$$\sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

The score for the wrong class must be at least 1 margin smaller than the score for the ground-truth class; Otherwise, there is a loss incurred

Deep Feedforward Neural Network with Sigmoid Hidden Units



Backpropagation with a Computational Graph



Train a Deep Neural Network with SGD

Split our training dataset into N mini-batches with batch size b

For Iteration = 1, …, Num_Max_Iterations

randomly choose a mini-batch D_i

$$
\begin{aligned}
v_{i+1} &:= 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i} \\
w_{i+1} &:= w_i + v_{i+1}
\end{aligned}
$$

where i is the iteration index, v is the momentum variable, ϵ is the learning rate, and $\left\langle \frac{\partial L}{\partial w}\big|_{w_i} \right\rangle_{D_i}$ is the average over the ith batch D_i of the derivative of the objective with respect to w, evaluated at w_i.

(you can also have two loops: outer loop over epochs, inner loop over mini-batches)

DNN works much worse than a shallow CNN even on MNIST!

~1.0% vs. ~0.60%

Why?

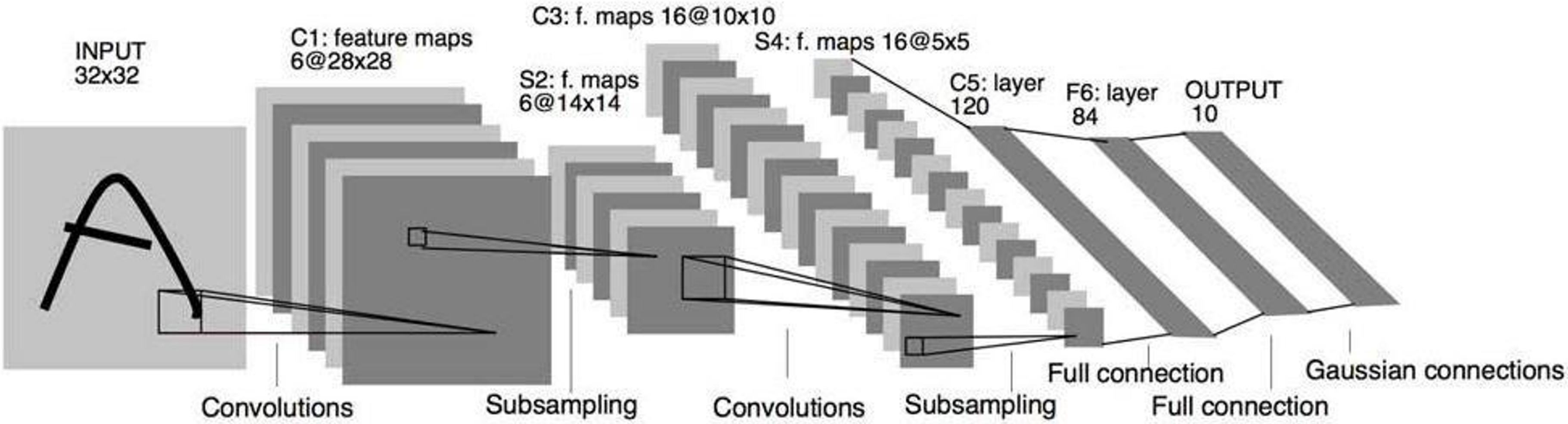
Hubel and Wiesel Experiment

https://www.youtube.com/watch?v=OGxVfKJqX5E

Message from Last Lecture

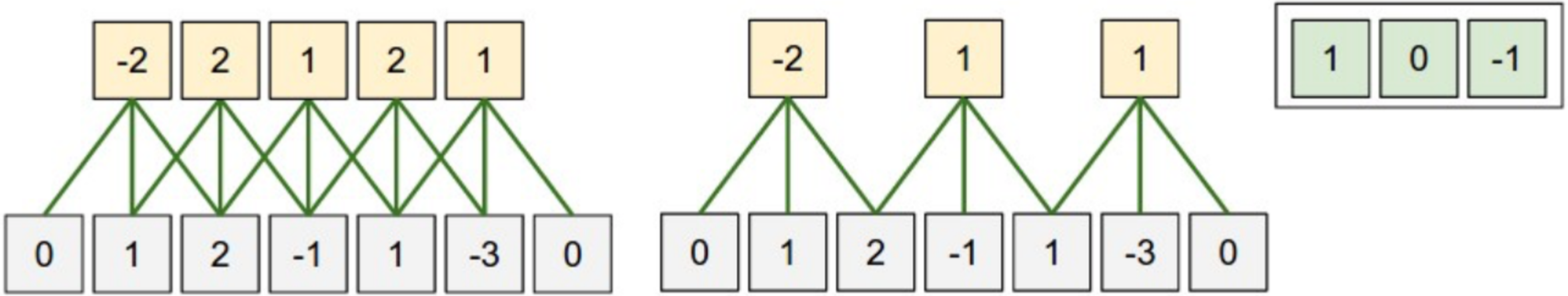
Deep learners should **combine** their **knowledge** with large-scale **data** to grow programs, **encode essential knowledge into network structures**, and let backpropagation and stochastic gradient descent do the heavy lifting.

Convolutional Neural Network: LeNet (1998)



LeCun et al., 1998

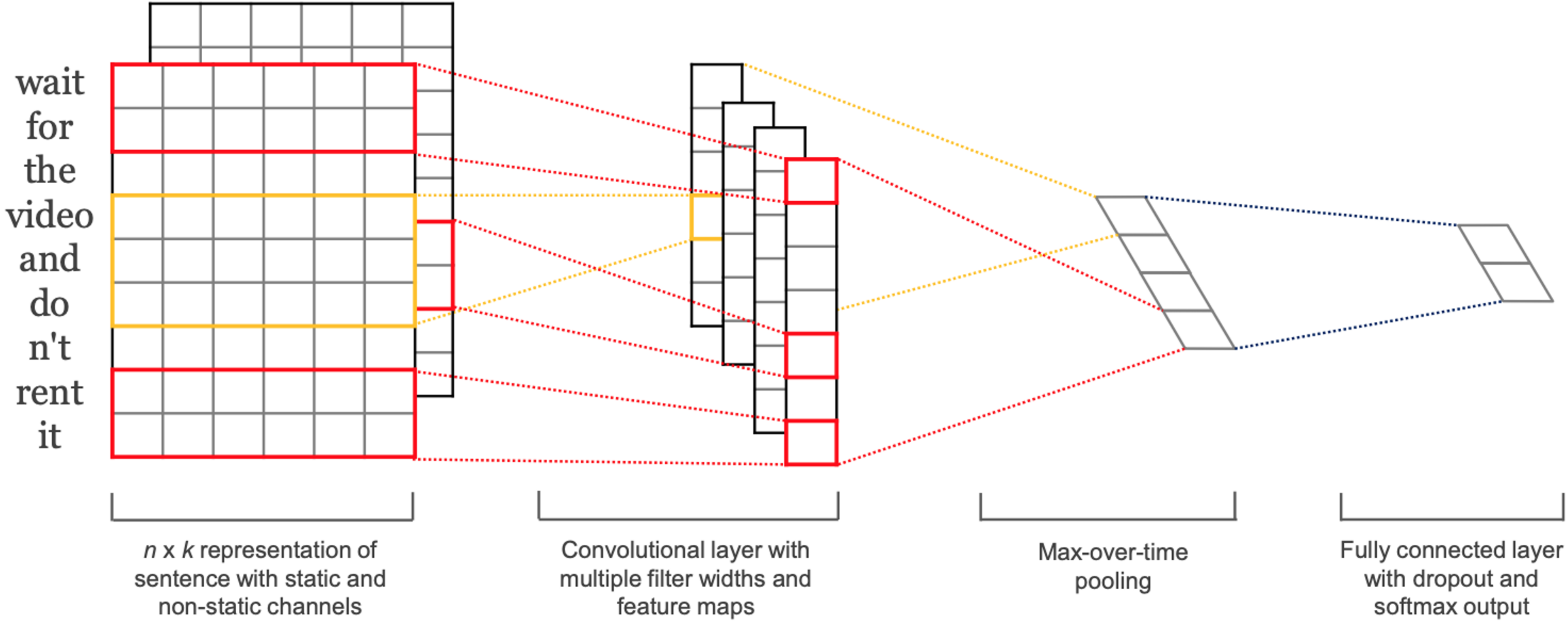
1D Convolution with W =5, F = 3, Stride = 2, Padding = 1



Output Size = $(W - F + 2P)/S + 1$

http://cs231n.github.io/convolutional-networks/

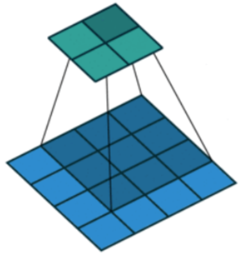
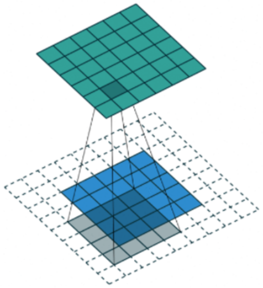
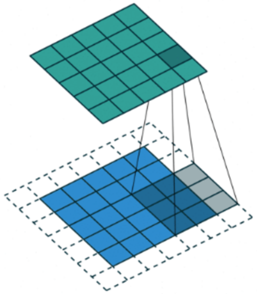
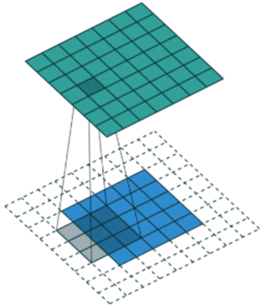
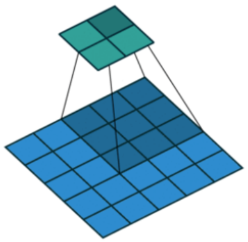
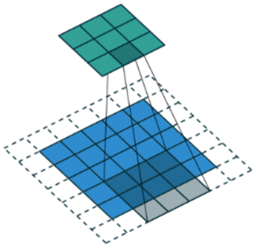
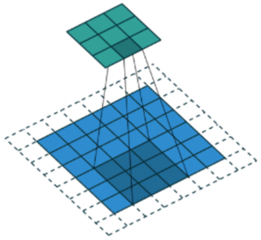
1D Convolution over Sentences



Yoon Kim, Convolutional Neural Networks for Sentence Classification. EMNLP 2014

2D Convolutions

N.B.: Blue maps are inputs, and cyan maps are outputs.

			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides
			
No padding, strides	Padding, strides	Padding, strides (odd)	

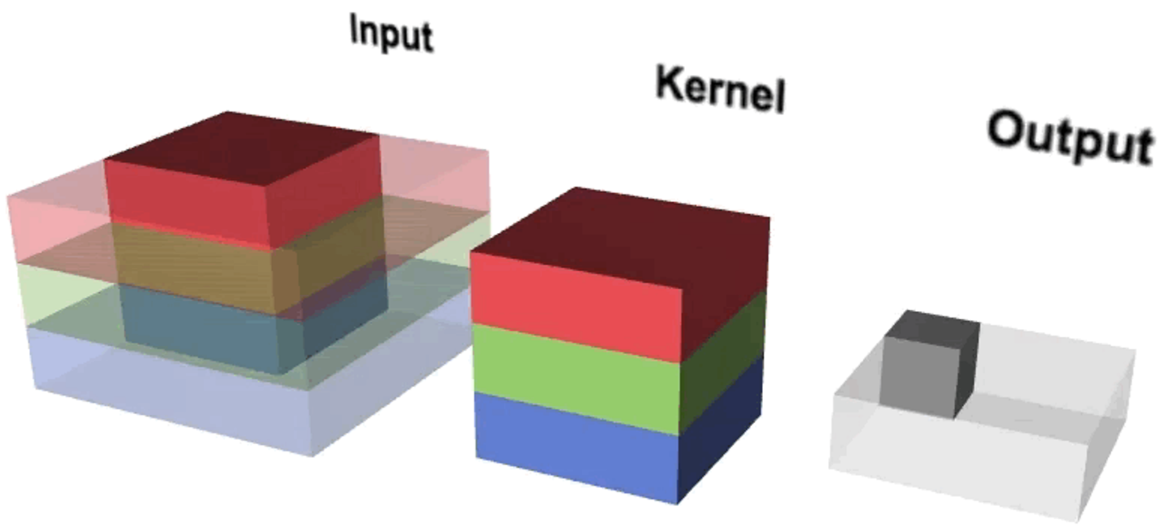
https://github.com/vdumoulin/conv_arithmetic

2D Convolution Animations

See the animation at

https://github.com/vdumoulin/conv_arithmetic

2D 3x3 Convolution Applied to RGB Input of Size 5x5

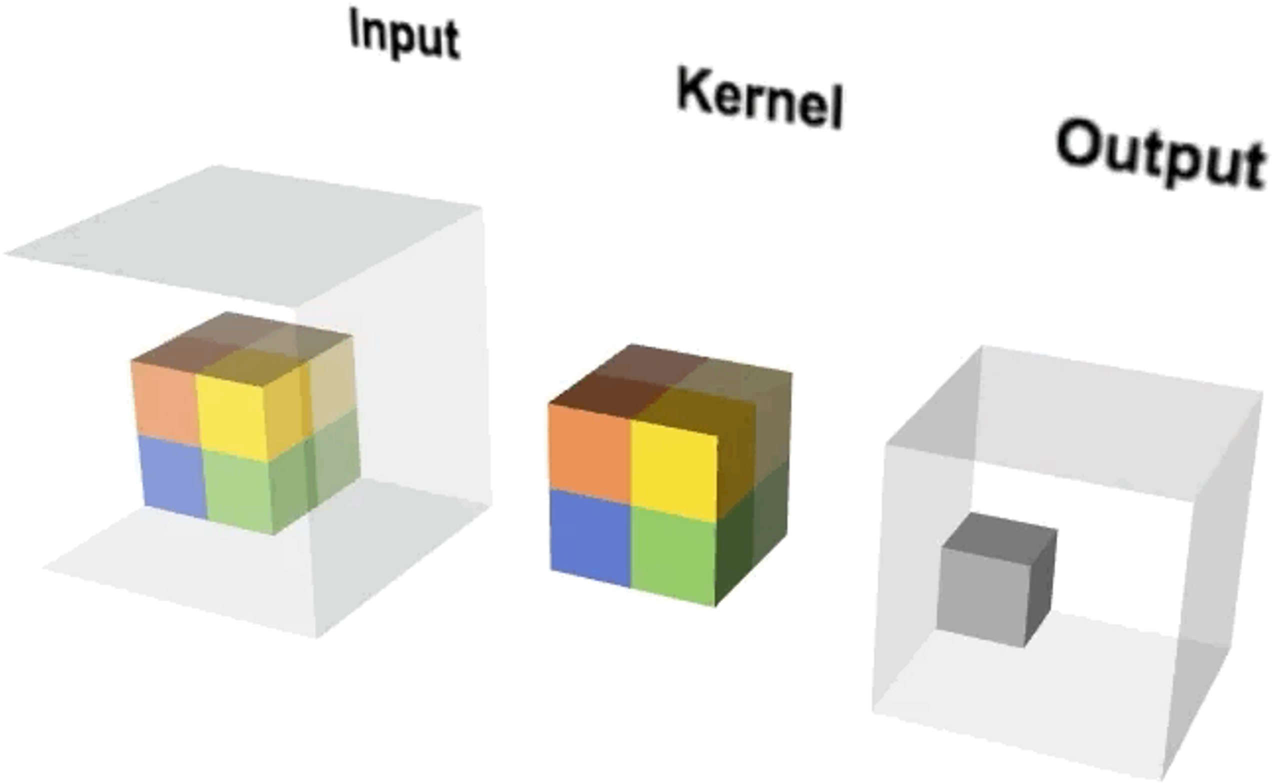


Picture credit: https://thomelane.github.io/convolutions/2DConvRGB.html

2D Convolutions in Numbers

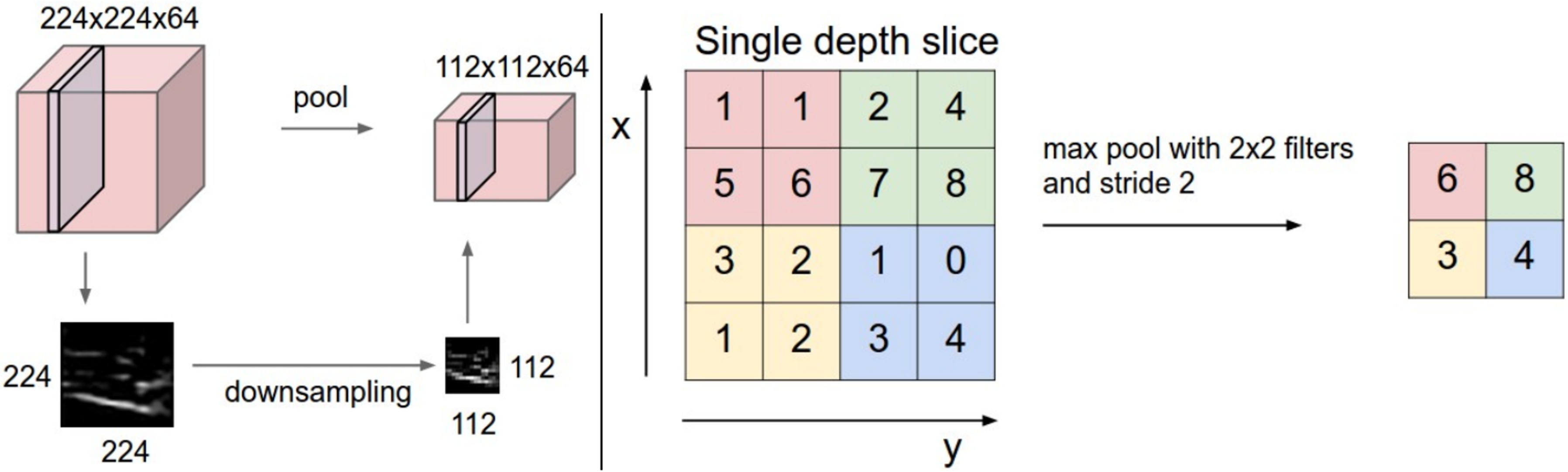
http://cs231n.github.io/convolutional-networks/

3D Convolution



Picture credit: https://thomelane.github.io/convolutions/3DConv.html

Max Pooling



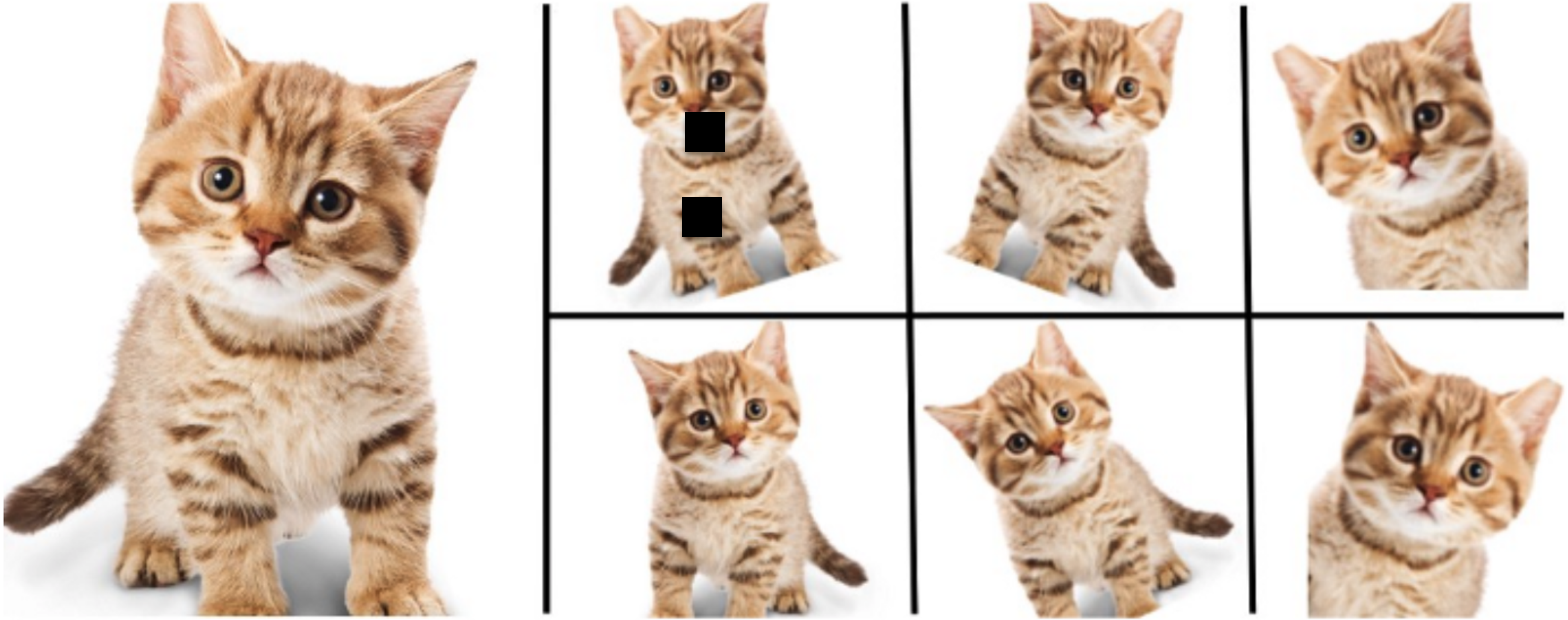
Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

http://cs231n.github.io/convolutional-networks/

Average Pooling is also widely used, especially in NLP

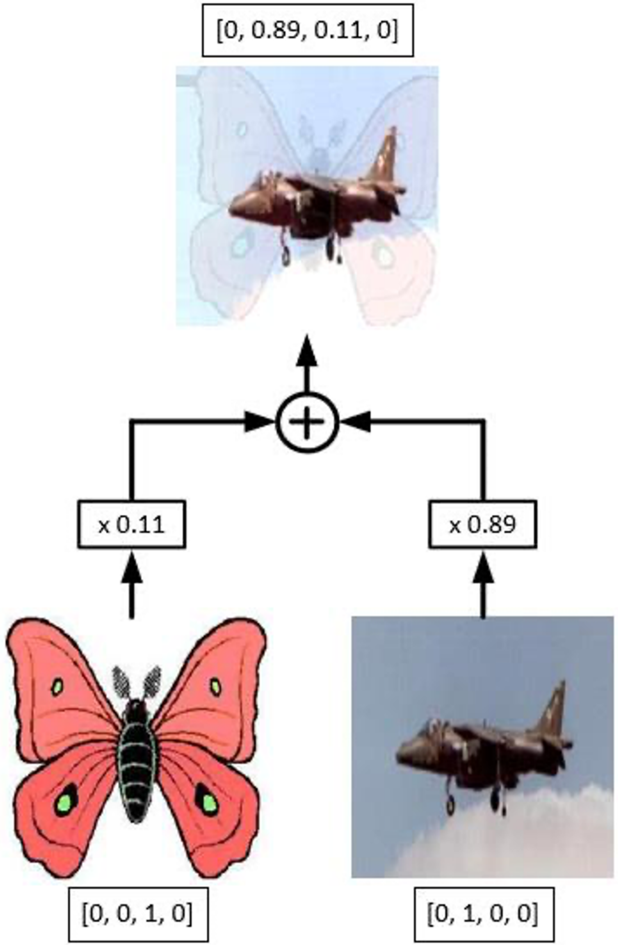
Data Augmentation

Random erasing, horizontal flipping, rotation, scaling (with cropping), cropping, contrast, color



Picture credit: https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

Mixup



$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j, \qquad \text{where } x_i, x_j \text{ are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j, \qquad \text{where } y_i, y_j \text{ are one-hot label encodings}$$

Zhang *et al*., Mixup: beyond empirical risk minimization. ICLR 2018.

Picture credit: https://www.dlology.com/blog/how-to-do-mixup-training-from-image-files-in-keras/

Case Study: AlexNet

[PDF] ImageNet Classification with Deep Convolutional Neural ...

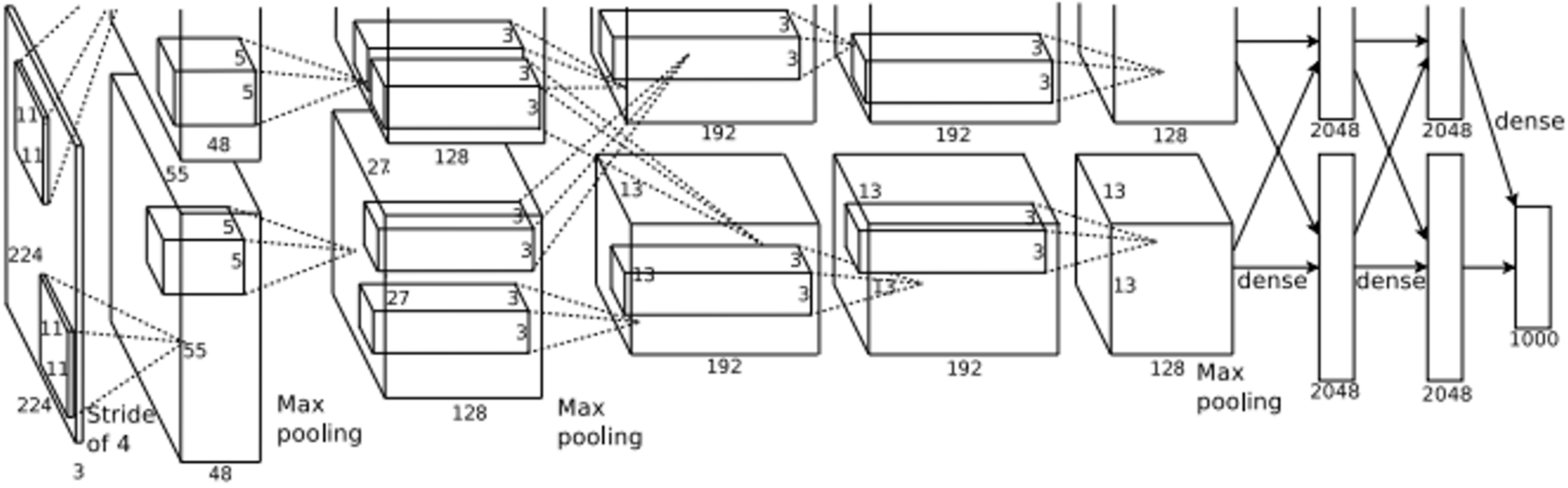
https://papers.nips.cc › paper › 4824-imagenet-classification-with-deep-co... ▾

by A Krizhevsky - 2012 - Cited by 54415 - Related articles

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 dif-.

NIPS 2012

AlexNet Network Structure



Pay attention to the output Size and the number of parameters

Training AlexNet using SGD with Momentum and Weight Decay

$$
\begin{aligned}
v_{i+1} &:= 0.9 \cdot v_i - 0.0005 \cdot \epsilon \cdot w_i - \epsilon \cdot \left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i} \\
w_{i+1} &:= w_i + v_{i+1}
\end{aligned}
$$

where i is the iteration index, v is the momentum variable, ϵ is the learning rate, and $\left\langle \frac{\partial L}{\partial w}\Big|_{w_i} \right\rangle_{D_i}$ is the average over the ith batch D_i of the derivative of the objective with respect to w, evaluated at w_i.

AlexNet with ReLU Converges Much Faster

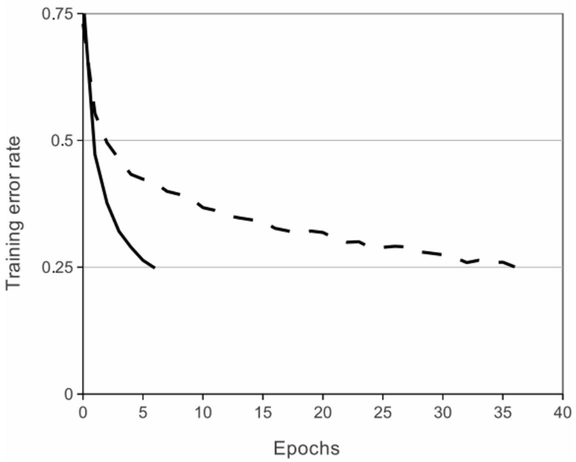


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

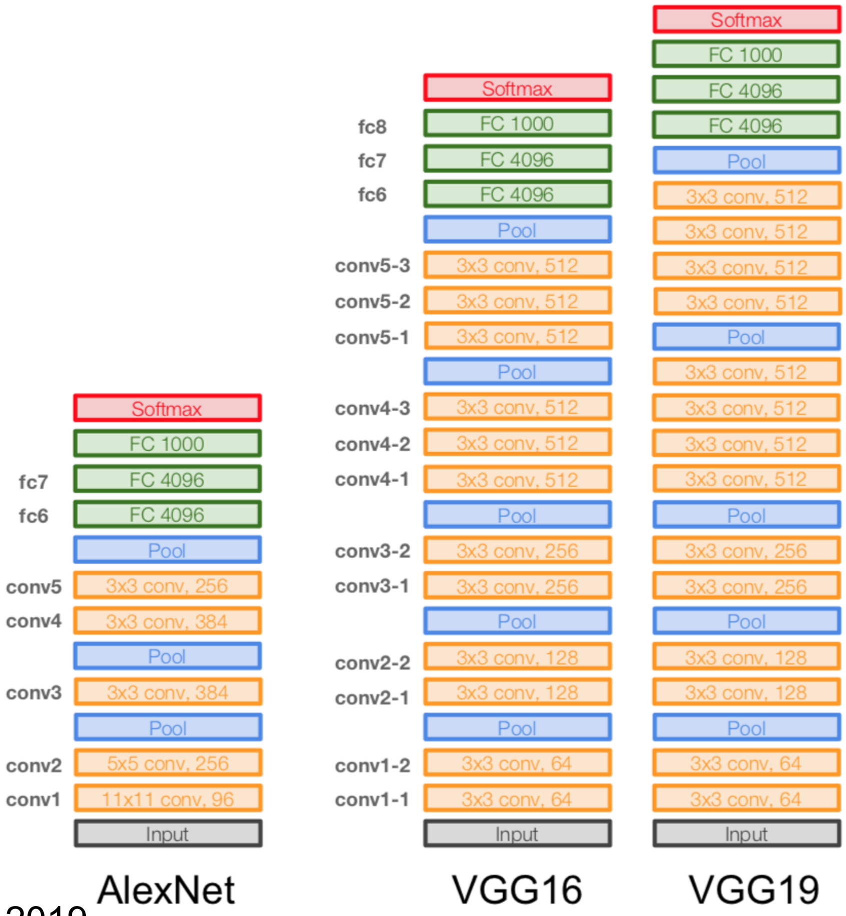
AlexNet vs. VGG

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



Picture Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

(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

Common names

TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2017

The deeper, the better?

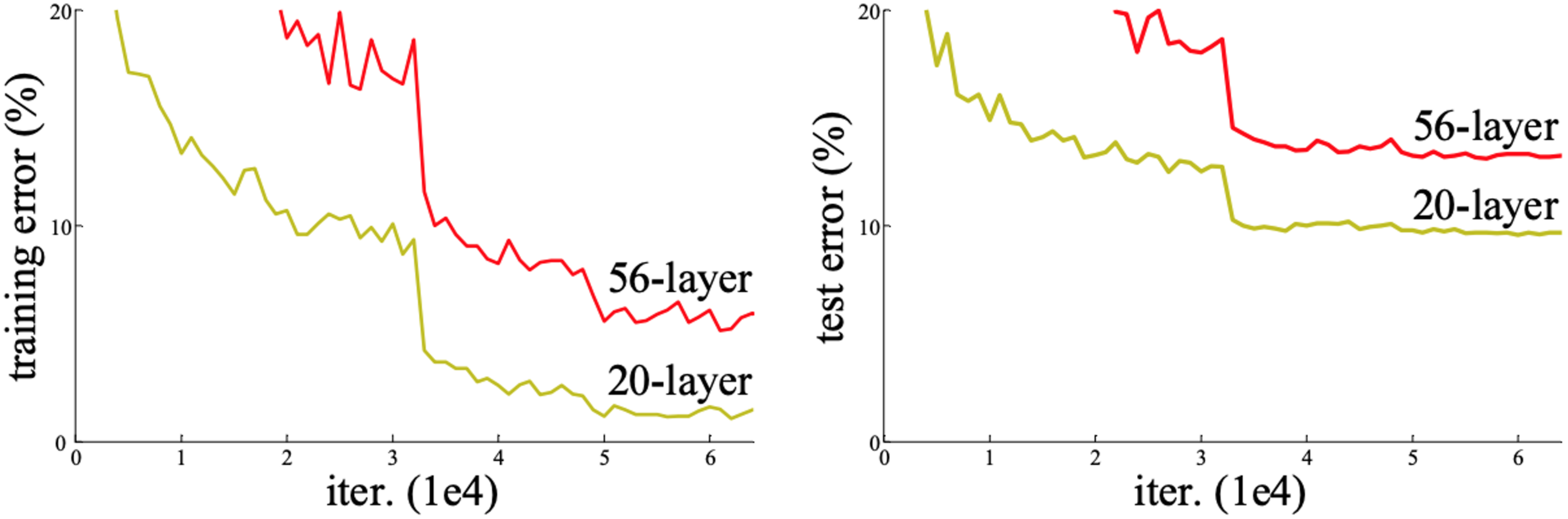


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet

He et al., CVPR 2015

Learning Residual Feature Maps is Easier

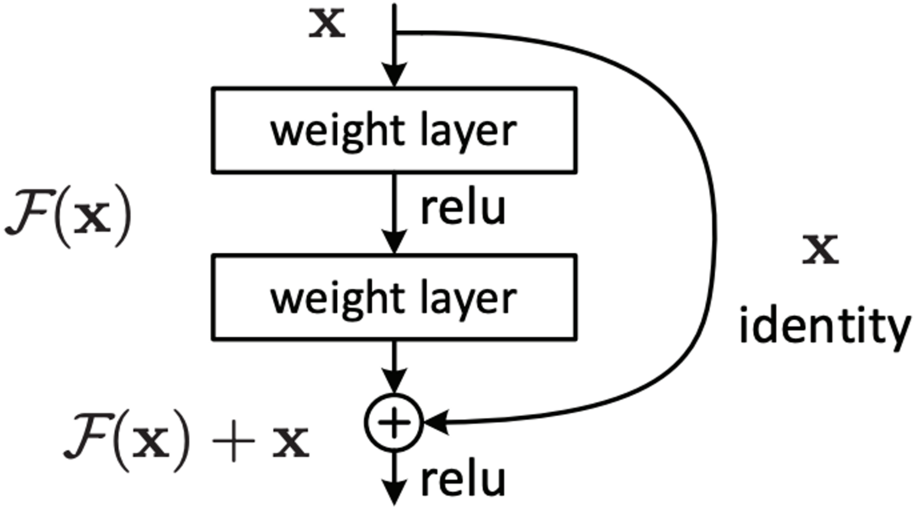
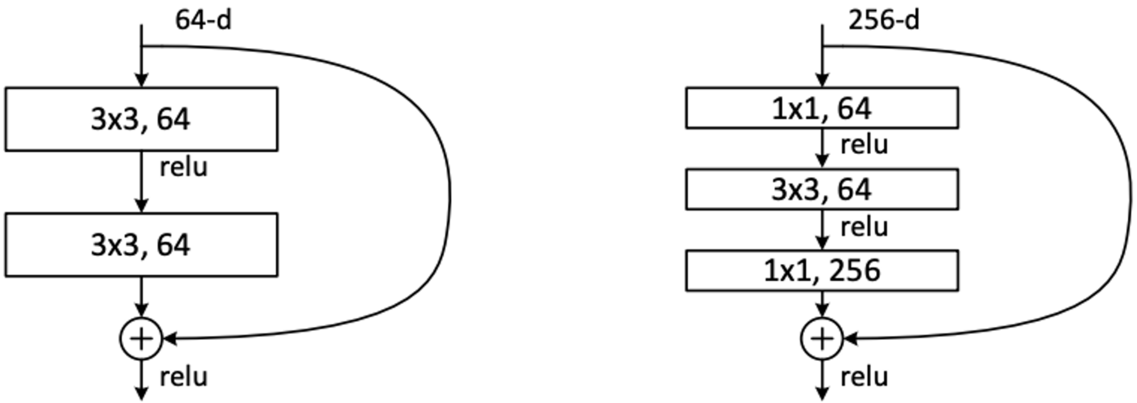


Figure 2. Residual learning: a building block.

He et al., Deep Residual Learning for Image Recognition. CVPR 2015

Learning Residual is Easier



He et al., Deep Residual Learning for Image Recognition. CVPR 2015

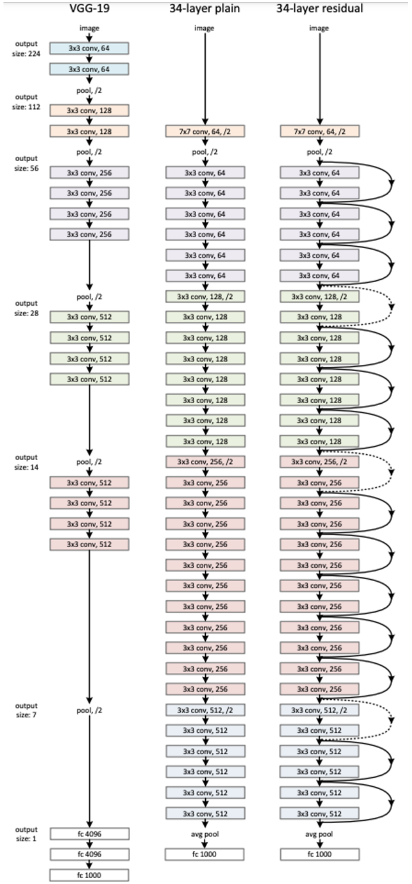
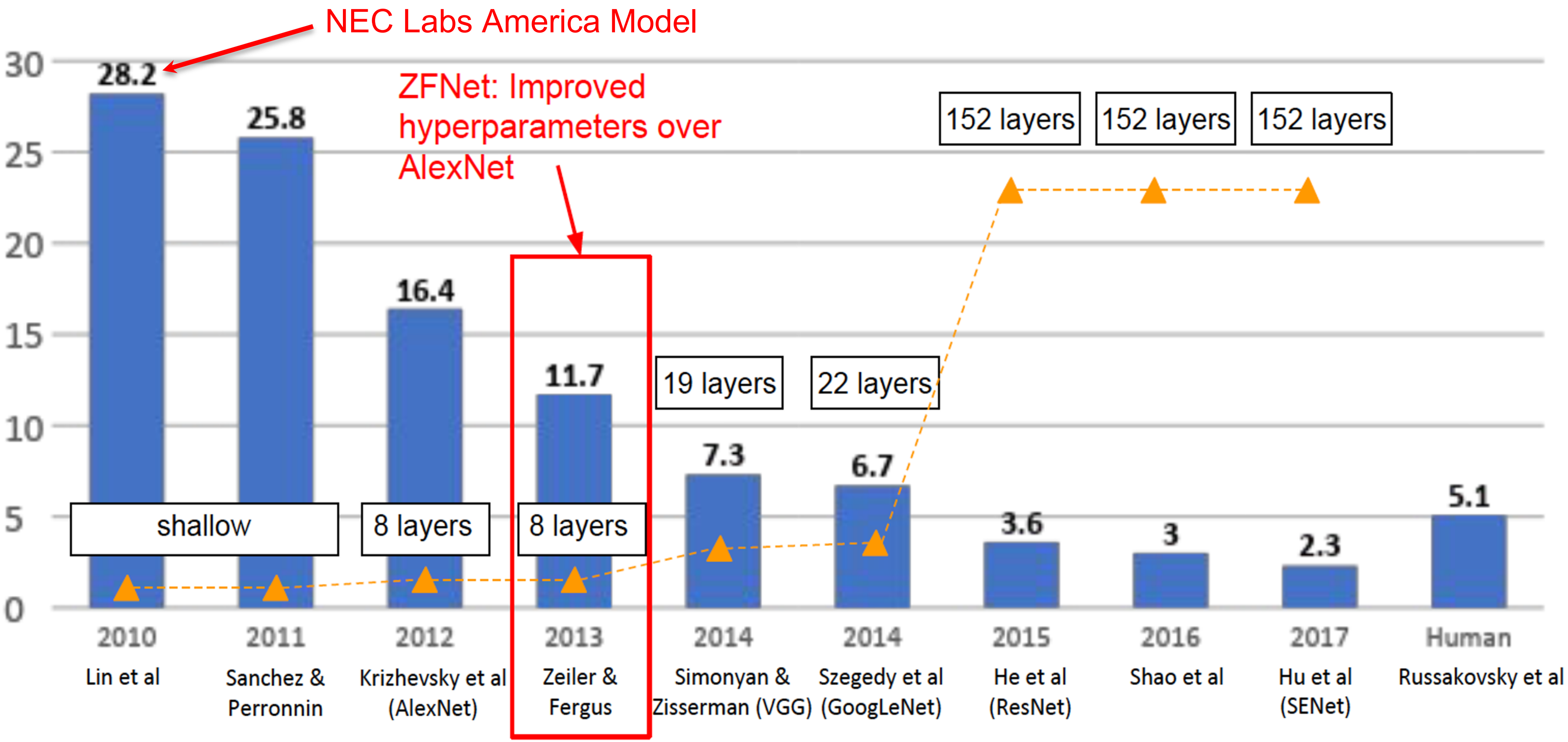


Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

VGG
vs.
ResNet

He et al., CVPR 2015

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Picture Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Conv2d in PyTorch

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,

Demonstration of training a simple CNN Classifier on CIFAR10 using PyTorch in Jupyter Notebook

Implement Your Own Forward and Backforward in PyTorch

```
import torch

class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

Implement Your Own Forward and Backforward in PyTorch

```
dtype = torch.float
device = torch.device("cpu")
# device = torch.device("cuda:0") # Uncomment this to run on GPU

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Create random Tensors for weights.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply
```

Implement Your Own Forward and Backforward in PyTorch

```
learning_rate = 1e-6
max_iter = 500
for t in range(max_iter):
    # To apply our Function, we use Function.apply method. We alias this as 'relu'.
    relu = MyReLU.apply

    # Forward pass: compute predicted y using operations; we compute
    # ReLU using our custom autograd operation.
    y_pred = relu(x.mm(w1)).mm(w2)

    # Compute and print loss
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass.
    loss.backward()

    # Update weights using gradient descent
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # Manually zero the gradients after updating weights
        w1.grad.zero_()
        w2.grad.zero_()
```

What can we do with a pre-trained Deep CNN on ImageNet?

- Simple Transfer learning
  - We transfer our learned model on the ImageNet to a different domain, for e.g., fine-grained flower category classification
  - It only works when the transferred domain is closely related to the source domain of ImageNet
- Few-shot learning
  - In this task, for each class, we only have a few labeled training examples
  - We can use the learned feature embeddings or their (weighted) mean as prototype(s)
- Zero-shot learning
  - In this task, we don't have any training example for some classes, but we have semantic descriptions about them
  - A simple idea: Output a 1000-class probabilities of a test image and use a convex combination of the semantic descriptions of the top k known classes to construct semantic features of the test image

Zero-shot Learning Example

Test Image	Softmax Baseline [7]	DeViSE [6]	ConSE (10)
	wig fur coat Saluki, gazelle hound Afghan hound, Afghan stole	water spaniel tea gown bridal gown, wedding gown spaniel tights, leotards	business suit **dress, frock** hairpiece, false hair, postiche swimsuit, swimwear, bathing suit kit, outfit
	ostrich, Struthio camelus black stork, Ciconia nigra vulture crane peacock	heron owl, bird of Minerva, bird of night hawk bird of prey, raptor, raptorial bird finch	**ratite, ratite bird, flightless bird** peafowl, bird of Juno common spoonbill New World vulture, cathartid Greek partridge, rock partridge
	sea lion plane, carpenter's plane cowboy boot loggerhead, loggerhead turtle goose	elephant turtle turtleneck, turtle, polo-neck flip-flop, thong handcart, pushcart, cart, go-cart	California sea lion **Steller sea lion** Australian sea lion South American sea lion eared seal
	hamster broccoli Pomeranian capuchin, ringtail weasel	**golden hamster, Syrian hamster** rhesus, rhesus monkey pipe shaker American mink, Mustela vison	**golden hamster, Syrian hamster** rodent, gnawer Eurasian hamster rhesus, rhesus monkey rabbit, coney, cony

https://arxiv.org/pdf/1312.5650.pdf

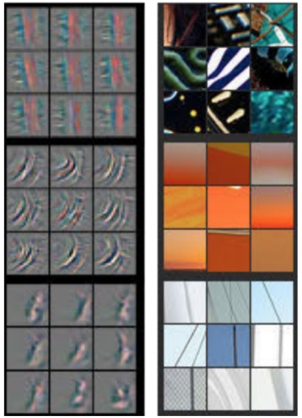
What do CNN (AlexNet-like) filters look like?

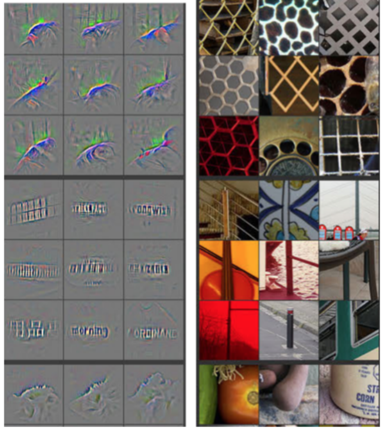
Zeiler and Fergus, 2013:
Visualizing and Understanding Convolutional Networks

An important convolutional operation called Transposed Convolution was invented in this paper, which will be discussed in Lec 5.

Layer 1

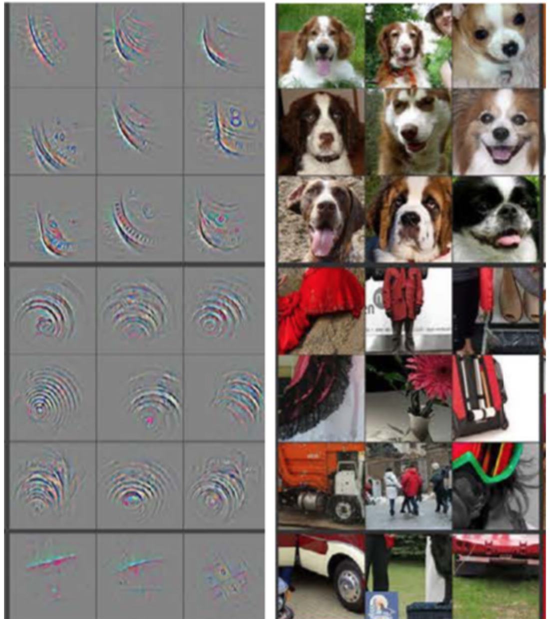
Layer 2

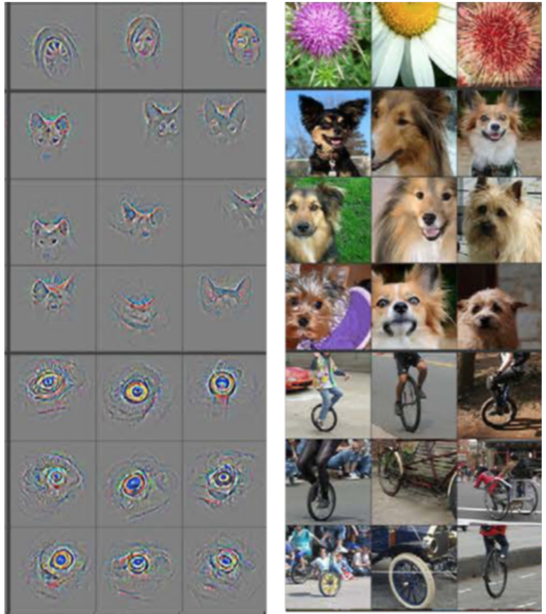




Layer 3

Layer 4





Layer 5

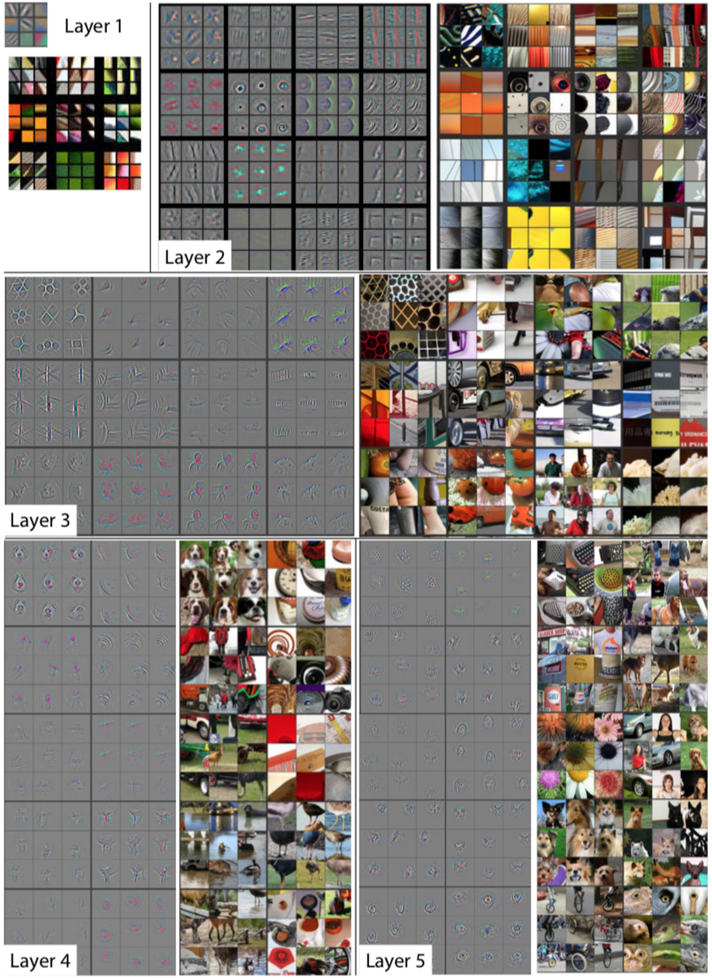
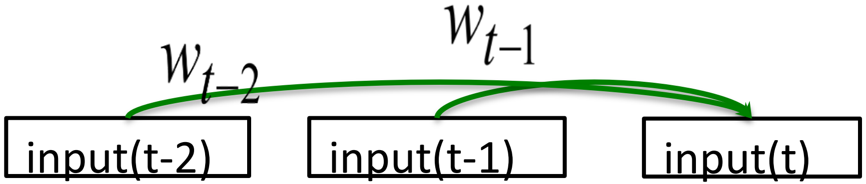


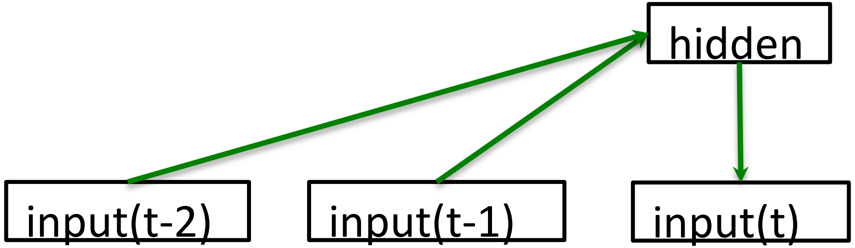
Figure 2. Visualization of features in a fully trained model. For layers 2-5 we show the top 9 activations in a random subset of feature maps across the validation data, projected down to pixel space using our deconvolutional network approach. Our reconstructions are *not* samples from the model: they are reconstructed patterns from the validation set that cause high activations in a given feature map. For each feature map we also show the corresponding image patches. Note: (i) the the strong grouping within each feature map, (ii) greater invariance at higher layers and (iii) exaggeration of

Memoryless models for sequences (Hinton's Slide)

- Autoregressive models
  Predict the next term in a sequence from a fixed number of previous terms using "delay taps".



- Feed-forward neural nets
  These generalize autoregressive models by using one or more layers of non-linear hidden units. *e.g.* Bengio's first language model.

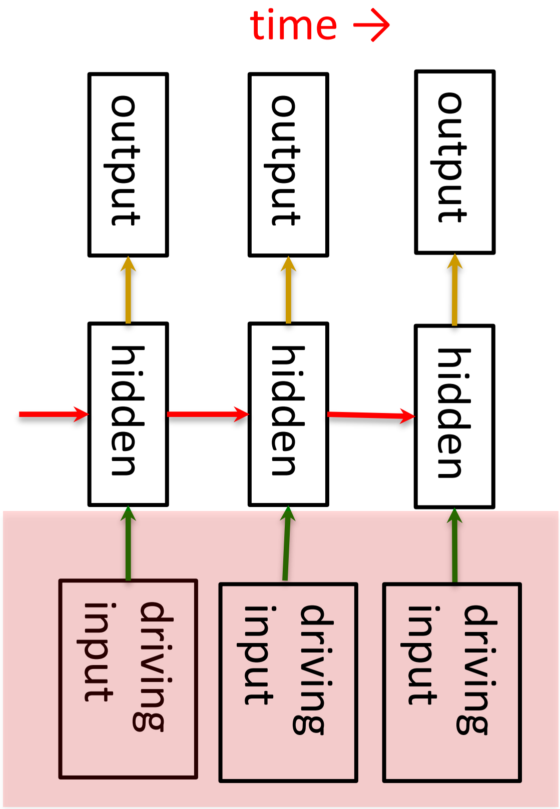


Beyond memoryless models (Hinton)

- If we give our generative model some hidden state, and if we give this hidden state its own internal dynamics, we get a much more interesting kind of model.
  - It can store information in its hidden state for a long time.
  - If the dynamics is noisy and the way it generates outputs from its hidden state is noisy, we can never know its exact hidden state.
  - The best we can do is to infer a probability distribution over the space of hidden state vectors.
- This inference is only tractable for two types of hidden state model.
  - The next three slides are mainly intended for people who already know about these two types of hidden state model. They show how RNNs differ.
  - Do not worry if you cannot follow the details.

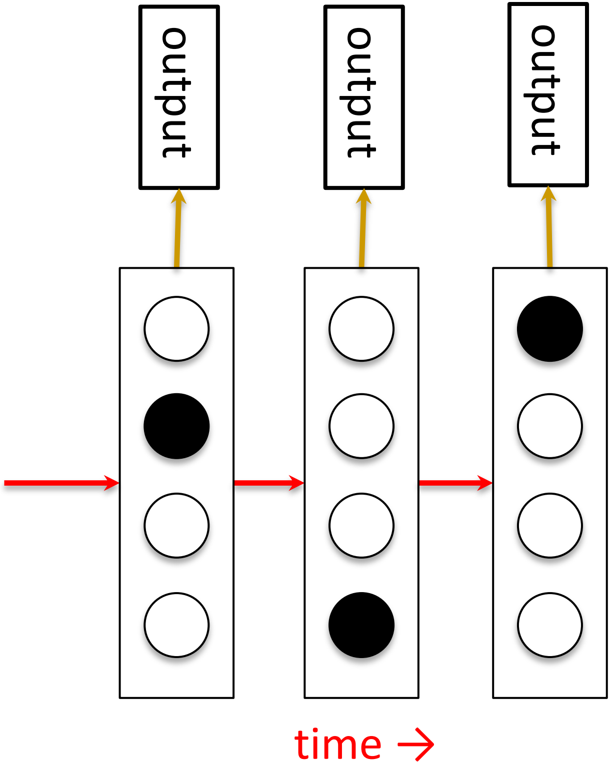
Linear Dynamical Systems (engineers love them!) (Hinton)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
  - The hidden state has linear dynamics with Gaussian noise and produces the observations using a linear model with Gaussian noise.
  - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
  - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian. It can be computed using "Kalman filtering".



Hidden Markov Models (computer scientists love them!) (Hinton)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
  - We cannot be sure which state produced a given output. So the state is “hidden”.
  - It is easy to represent a probability distribution across N states with N numbers.
- To predict the next output we need to infer the probability distribution over hidden states.
  - HMMs have efficient algorithms for inference and learning.

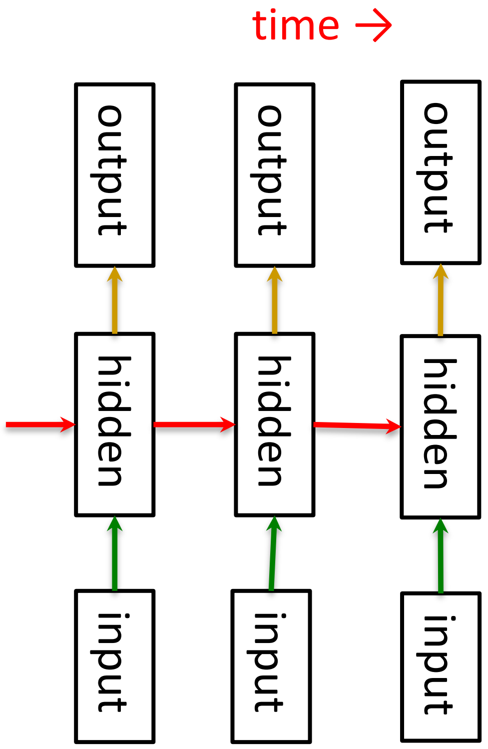


A fundamental limitation of HMMs (Hinton)

- Consider what happens when a hidden Markov model generates data.
  - At each time step it must select one of its hidden states. So with N hidden states it can only remember log(N) bits about what it generated so far.
- Consider the information that the first half of an utterance contains about the second half:
  - The syntax needs to fit (e.g. number and tense agreement).
  - The semantics needs to fit. The intonation needs to fit.
  - The accent, rate, volume, and vocal tract characteristics must all fit.
- All these aspects combined could be 100 bits of information that the first half of an utterance needs to convey to the second half. 2^100 is big!

Recurrent neural networks (Hinton)

- RNNs are very powerful, because they combine two properties:
  - Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.

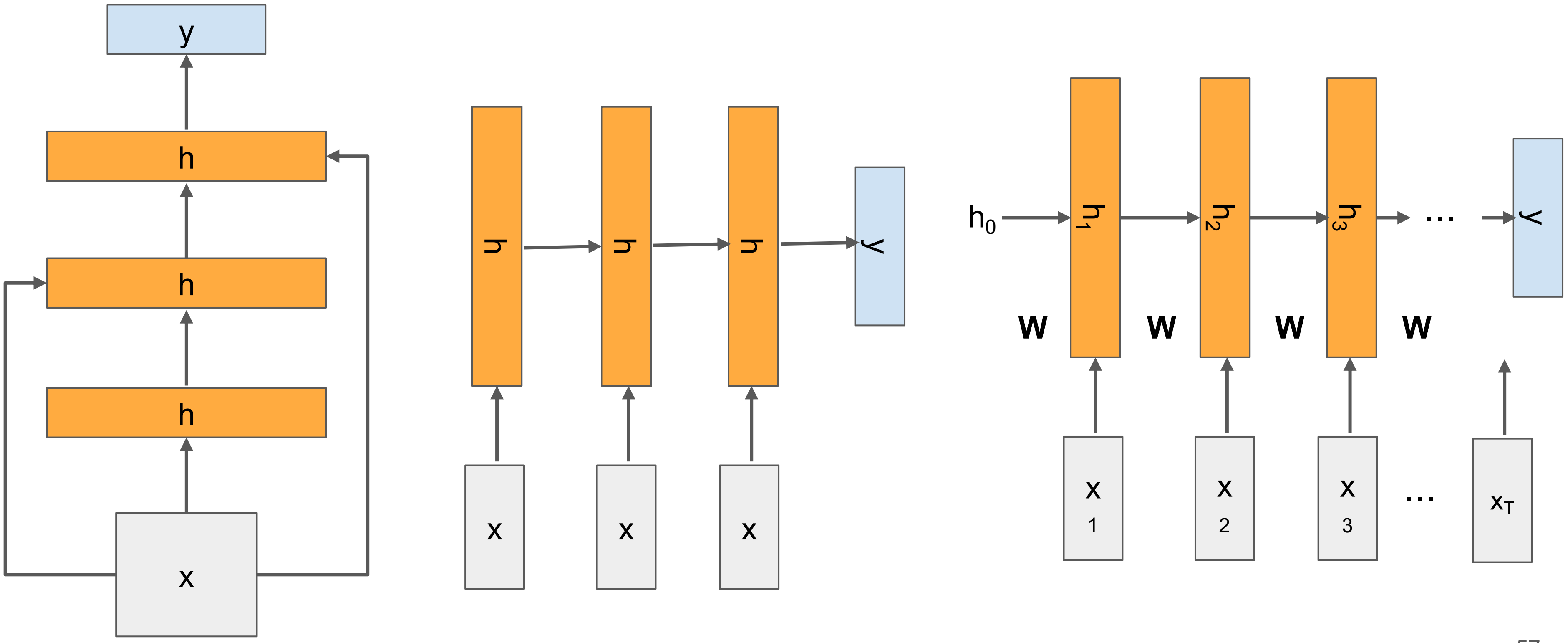


Do generative models need to be stochastic? (Hinton)

- Linear dynamical systems and hidden Markov models are stochastic models.
  - But the posterior probability distribution over their hidden states given the observed data so far is a deterministic function of the data.
- Recurrent neural networks are deterministic.
  - So think of the hidden state of an RNN as the equivalent of the deterministic probability distribution over hidden states in a linear dynamical system or hidden Markov model.

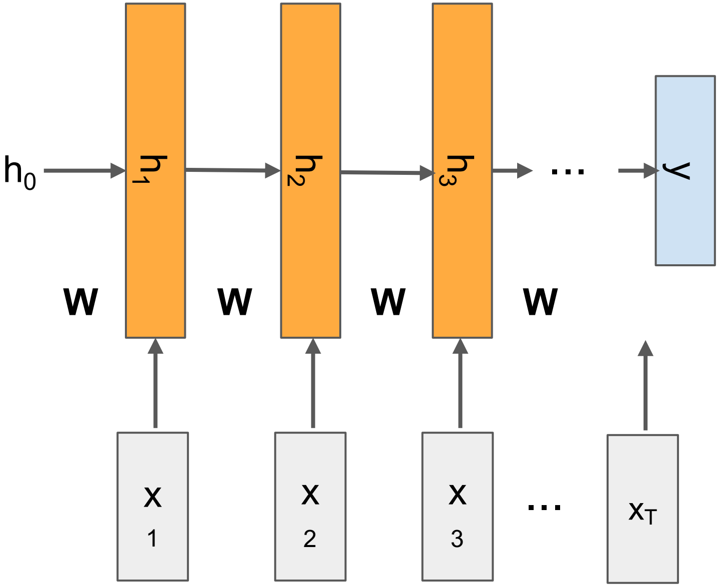
From Standard Neural Networks to Recurrent Neural Networks

Let's make the model easily extendable to model sequences with arbitrary lengths by weight sharing



Recurrent Neural Networks (RNN)

At time step t, the hidden units accumulate past information about the input sequence. Hidden activity vector h_t only depend on current input x_t and previous hidden activity vector h_{t-1}



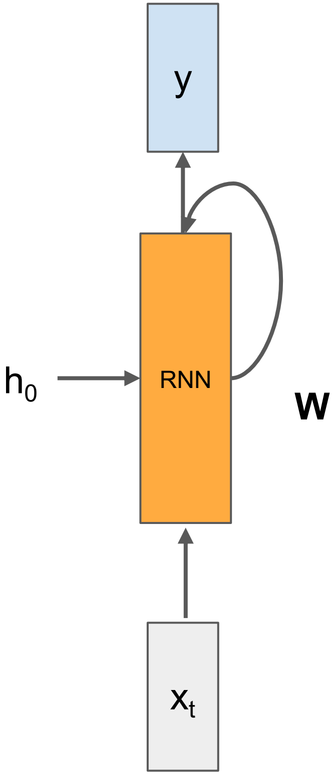
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

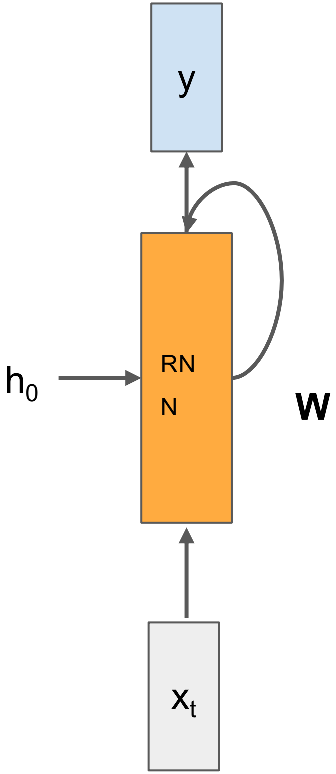
old state

input vector at
some time step

some function
with parameters W



Vanilla Recurrent Neural Networks

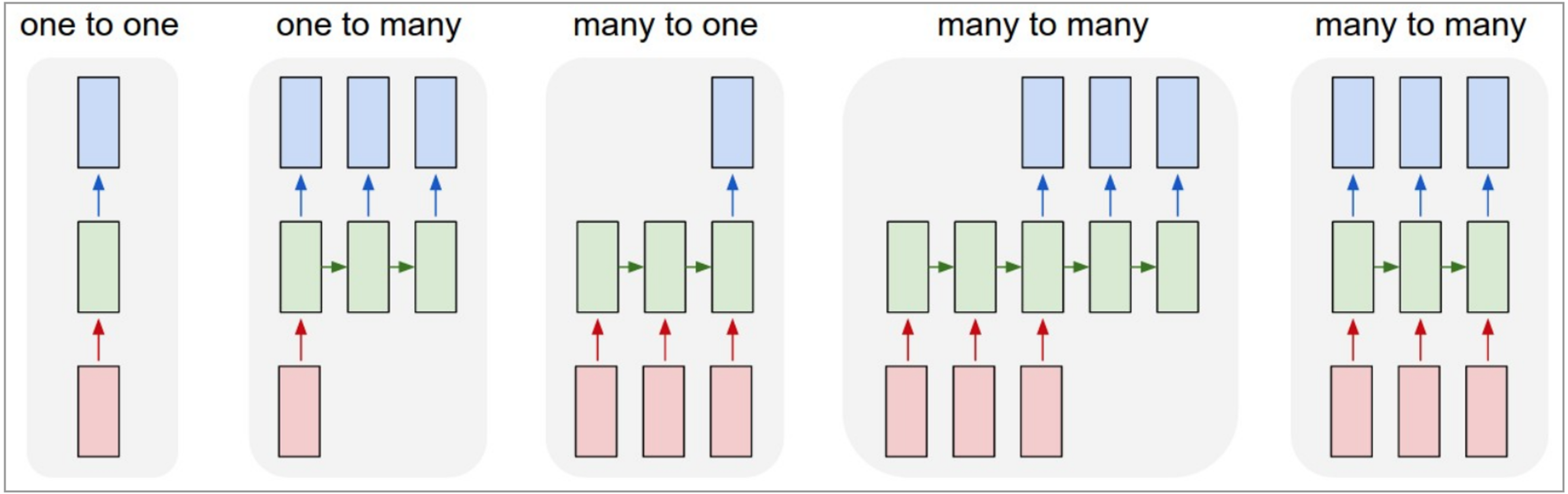


$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

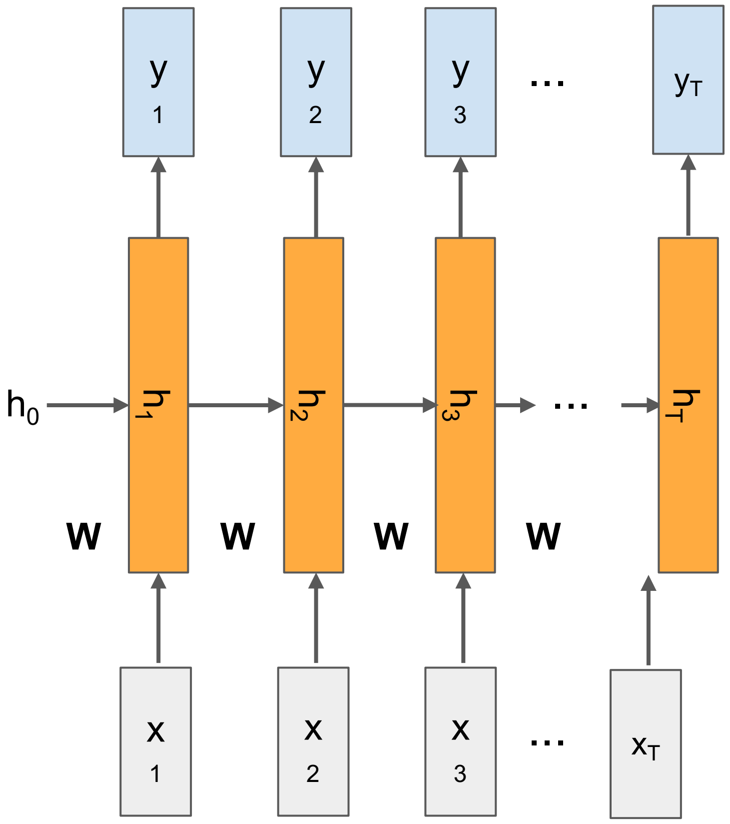
Different Architectures of RNN



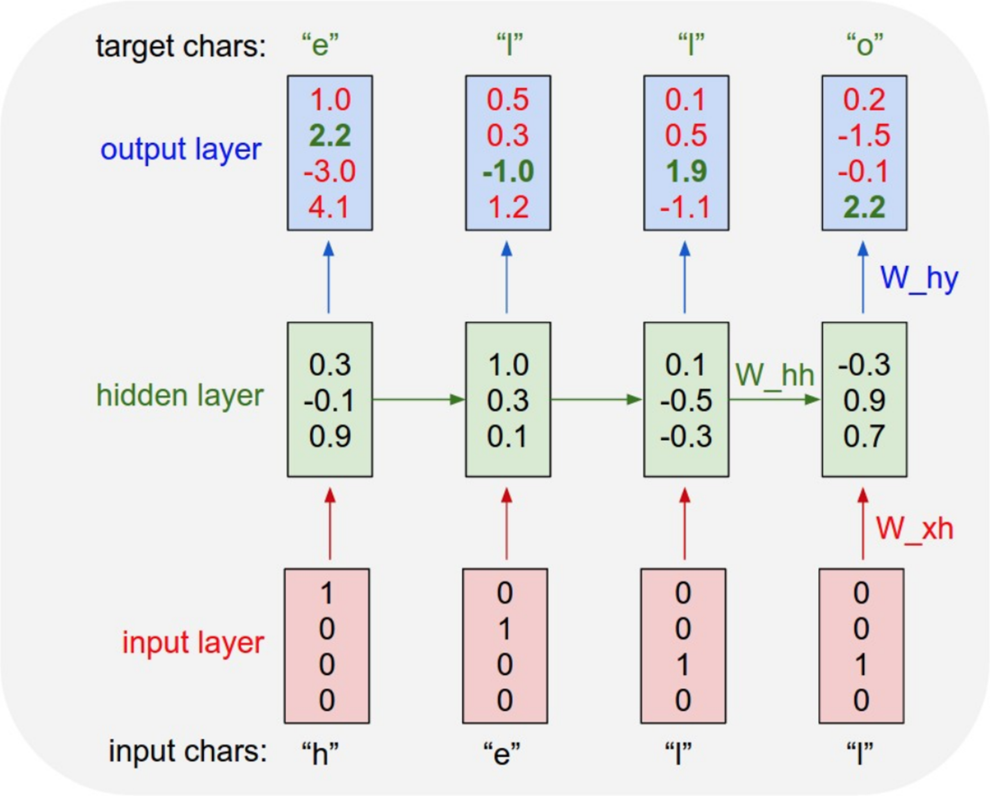
Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

Picture Credit: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Many-to-Many Vanilla RNN



Training of Char-RNN



$$\Pr(\mathbf{x}) = \prod_{t=1}^{T} \Pr(x_{t+1}|y_t)$$

$$\hat{y}_t = b_y + \sum_{n=1}^{N} W_{h^n y} h_t^n$$

$$y_t = \mathcal{Y}(\hat{y}_t)$$

$$\Pr(x_{t+1} = k|y_t) = y_t^k = \frac{\exp\left(\hat{y}_t^k\right)}{\sum_{k'=1}^{K} \exp\left(\hat{y}_t^{k'}\right)}$$

$$\mathcal{L}(\mathbf{x}) = -\sum_{t=1}^{T} \log y_t^{x_{t+1}}$$

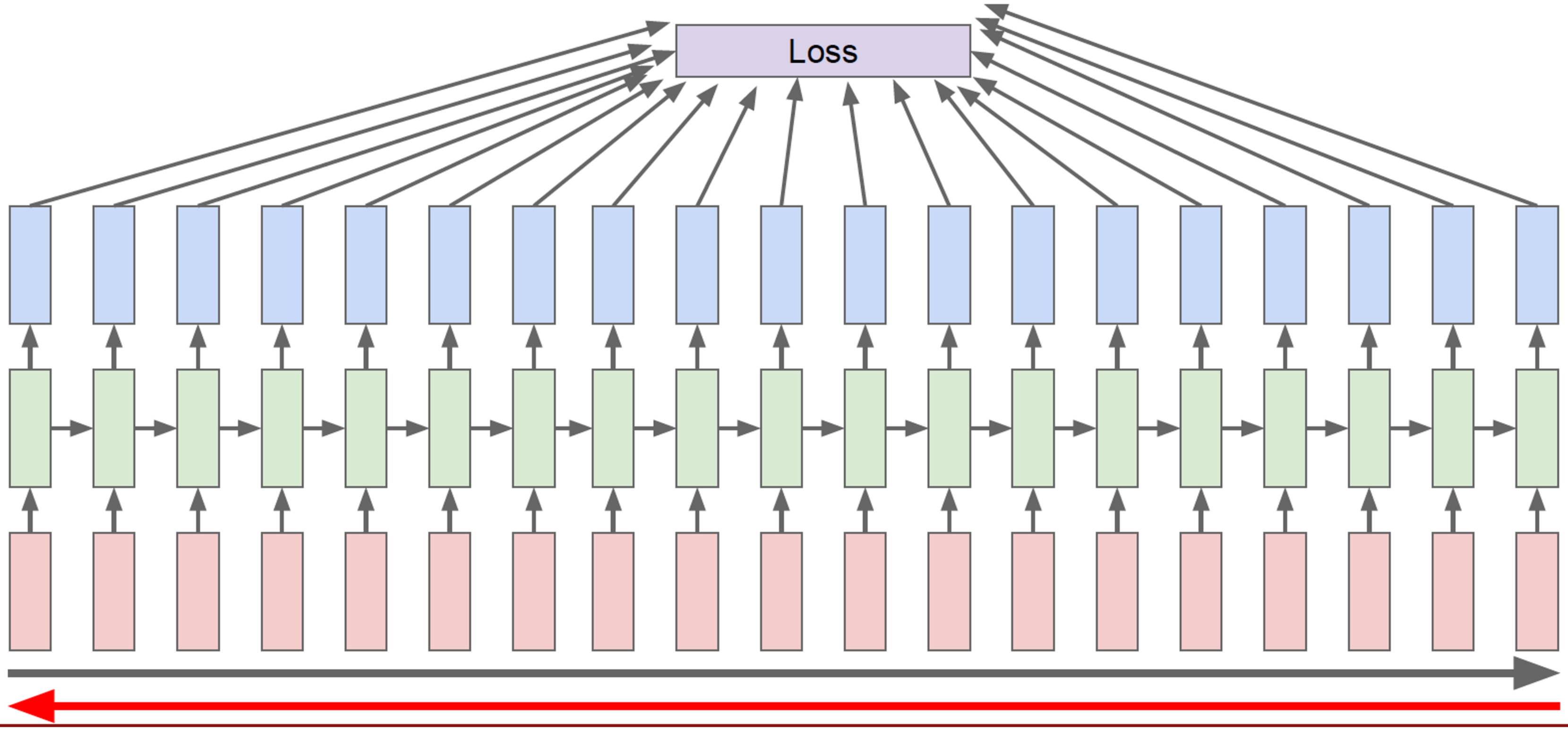
$$\implies \frac{\partial \mathcal{L}(\mathbf{x})}{\partial \hat{y}_t^k} = y_t^k - \delta_{k,x_{t+1}}$$

An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

Picture Credit: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

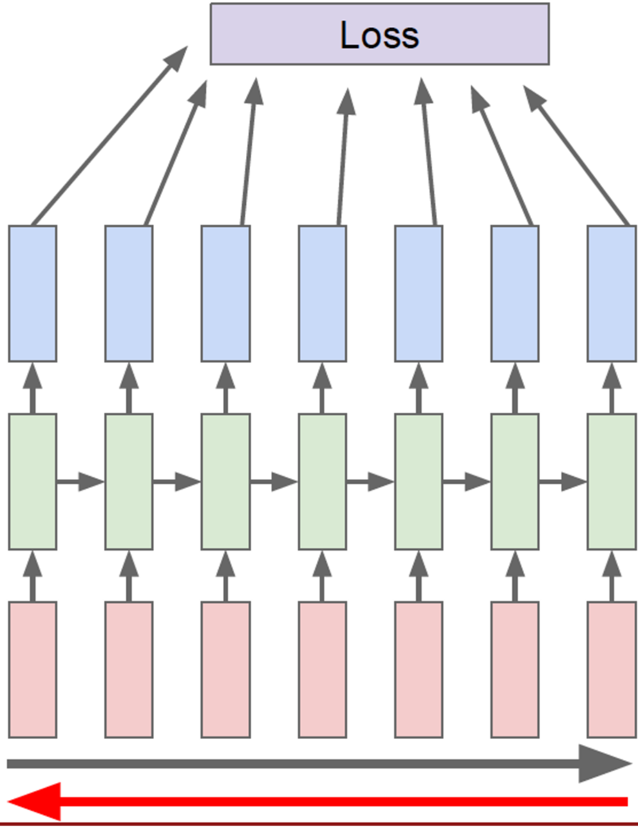
Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient



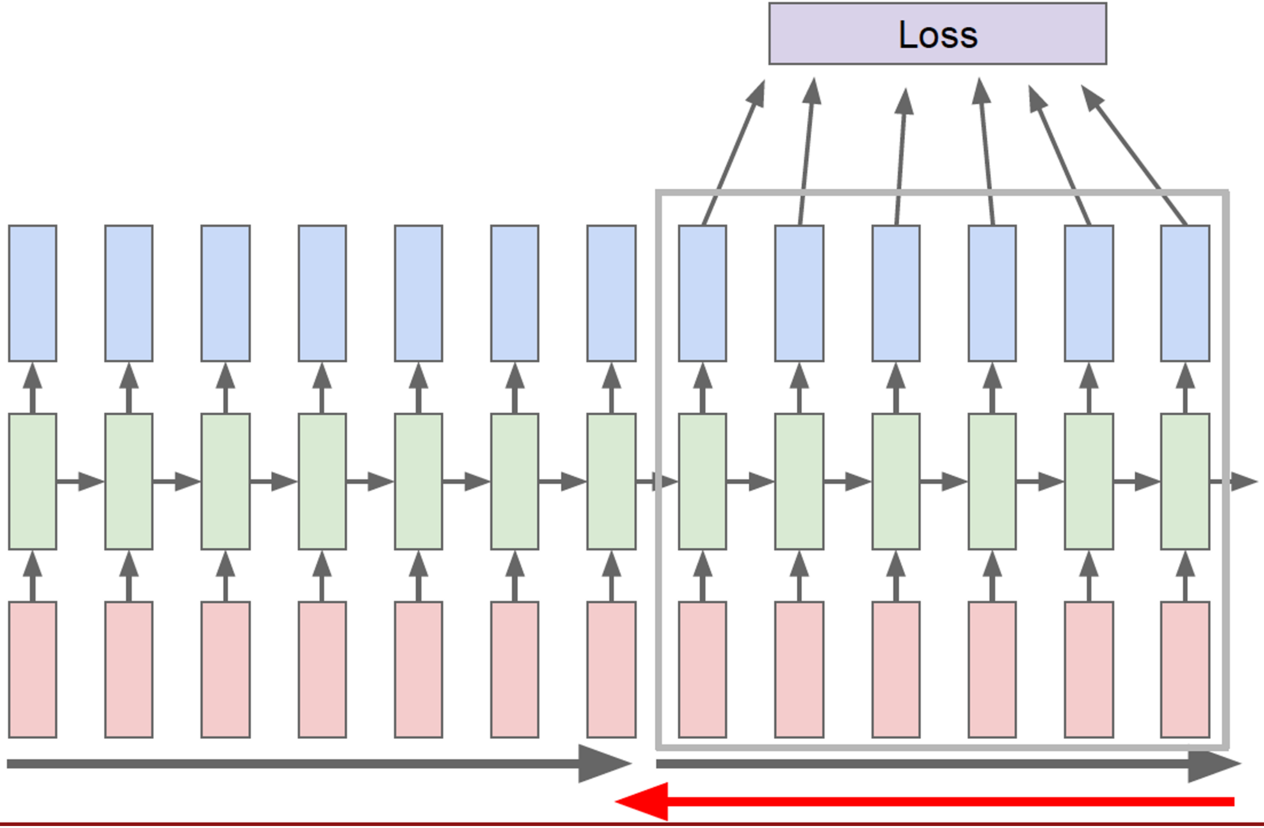
Slide Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Truncated Backpropagation through time



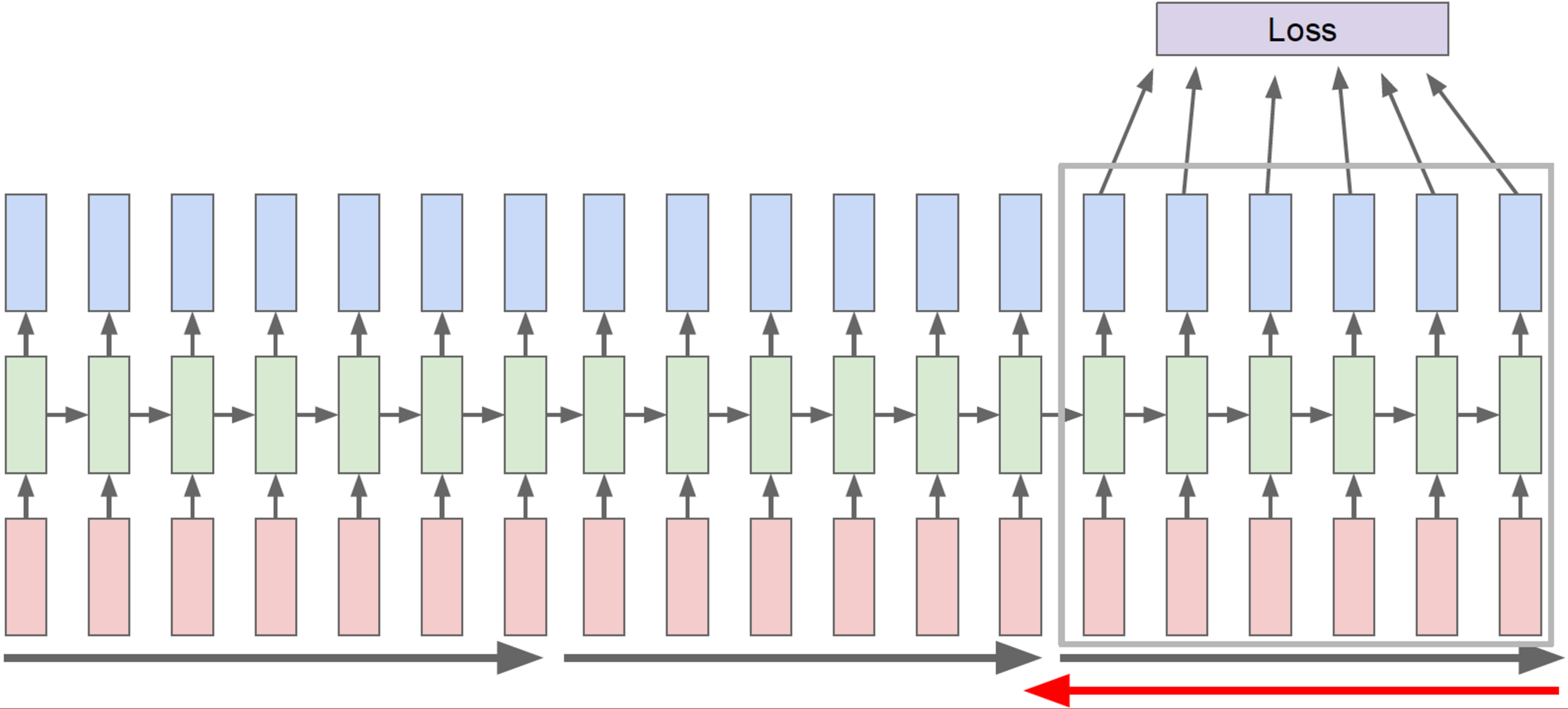
Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation through time



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

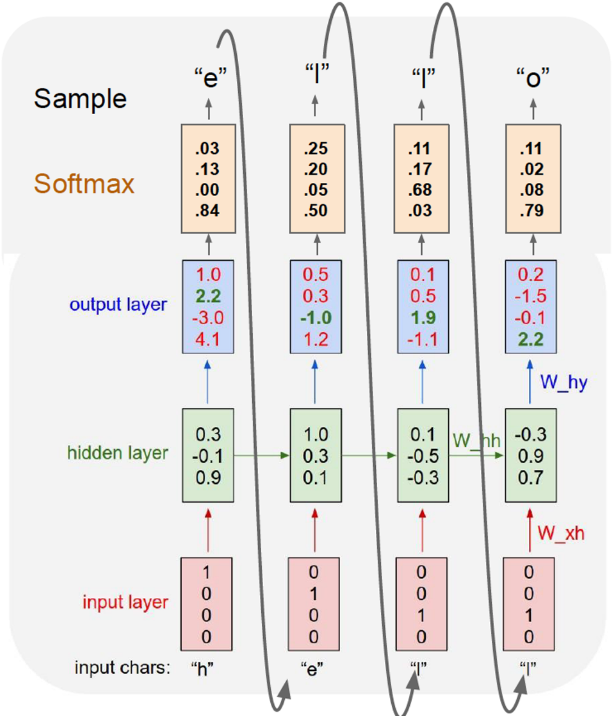
Truncated Backpropagation through time



Slide Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Inference of Char-RNN

At test time, sample a character from the current model at each step, feed the current sampled character as input to the next time step



Slide Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Karpathy’s Char-RNN on Shakespeare Articles

```
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

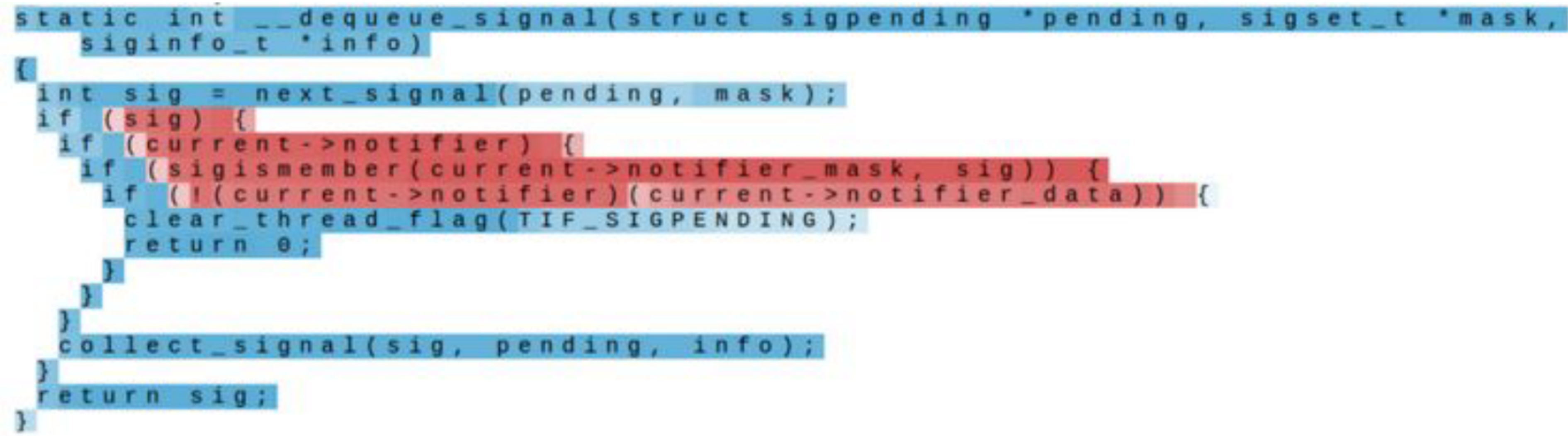
Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

```c
static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

Generated C code

Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Searching for interpretable cells



if statement cell

Karpathy, Johnson, and Fei-Fei: Visualizing and Understanding Recurrent Networks, ICLR Workshop 2016

Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

https://gist.github.com/karpathy/d4dee566867f8291f086

min-char-rnn.py

```python
"""
Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
BSD License
"""
import numpy as np

# data I/O
data = open('input.txt', 'r').read() # should be simple plain text file
chars = list(set(data))
data_size, vocab_size = len(data), len(chars)
print 'data has %d characters, %d unique.' % (data_size, vocab_size)
char_to_ix = { ch:i for i,ch in enumerate(chars) }
ix_to_char = { i:ch for i,ch in enumerate(chars) }

# hyperparameters
hidden_size = 100 # size of hidden layer of neurons
seq_length = 25 # number of steps to unroll the RNN for
learning_rate = 1e-1

# model parameters
Wxh = np.random.randn(hidden_size, vocab_size)*0.01 # input to hidden
Whh = np.random.randn(hidden_size, hidden_size)*0.01 # hidden to hidden
Why = np.random.randn(vocab_size, hidden_size)*0.01 # hidden to output
bh = np.zeros((hidden_size, 1)) # hidden bias
by = np.zeros((vocab_size, 1)) # output bias

def lossFun(inputs, targets, hprev):
  """
  inputs,targets are both list of integers.
  hprev is Hx1 array of initial hidden state
  returns the loss, gradients on model parameters, and last hidden state
  """
```

Why Vanishing and Exploding Gradient of Vanilla RNN Happens

$$h_t = f_W(h_{t-1}, x_t)$$

$$\downarrow$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Suppose we are using a many-to-many RNN for sequence labeling

$$\mathcal{E}_t = \mathcal{L}(\mathbf{h}_t) \quad \mathcal{E} = \sum_t \mathcal{E}_t$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k^+}{\partial \theta} \right)$$

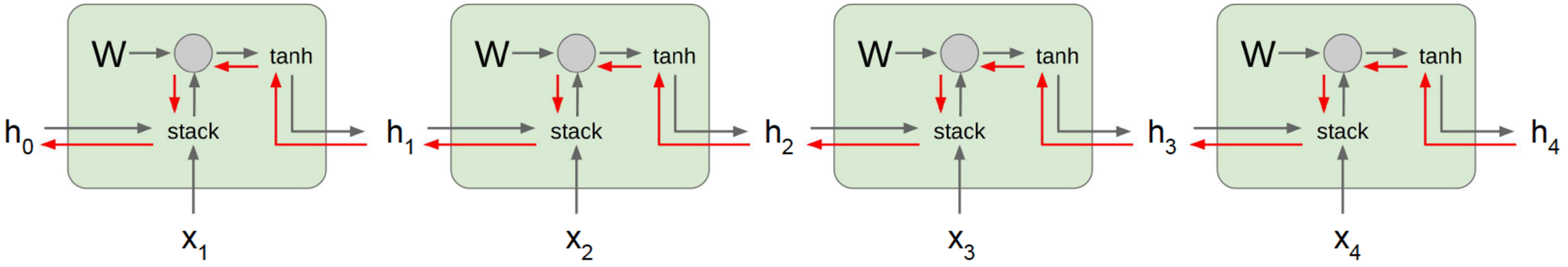
$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \Pi_{t \ge i > k} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} = \Pi_{t \ge i > k} W_{hh}^T diag(tanh'(W_{hh}h_{i-1} + W_{xh}x_i))$$

$\dfrac{\partial \mathbf{h}_k^+}{\partial \theta}$ is the immediate partial derivative of hidden activity vector with respect to network weights

Pascanu et al., On the difficulty of training recurrent neural networks. ICML 2013

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Largest singular value > 1:
Exploding gradients

Largest singular value < 1:
Vanishing gradients

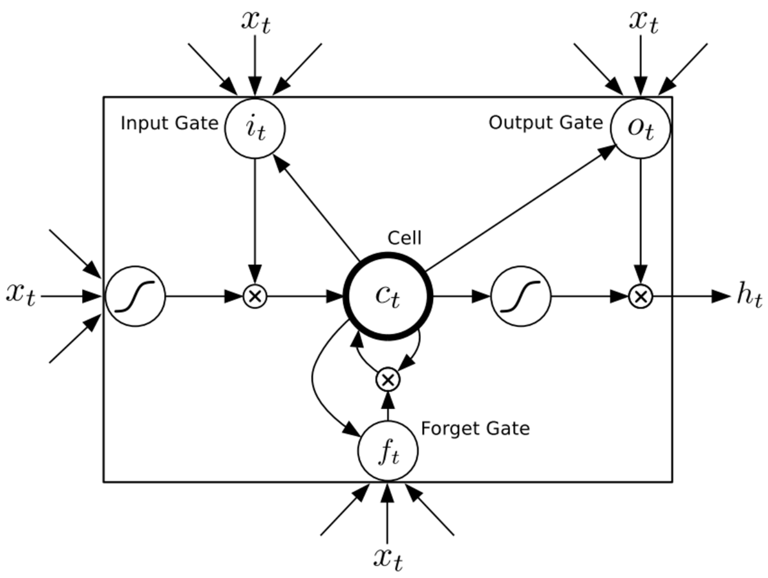
Gradient norm Clipping

Algorithm 1 Pseudo-code for norm clipping

$\hat{\mathbf{g}} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta}$
if $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
  $\hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|} \hat{\mathbf{g}}$
end if

Design a better architecture

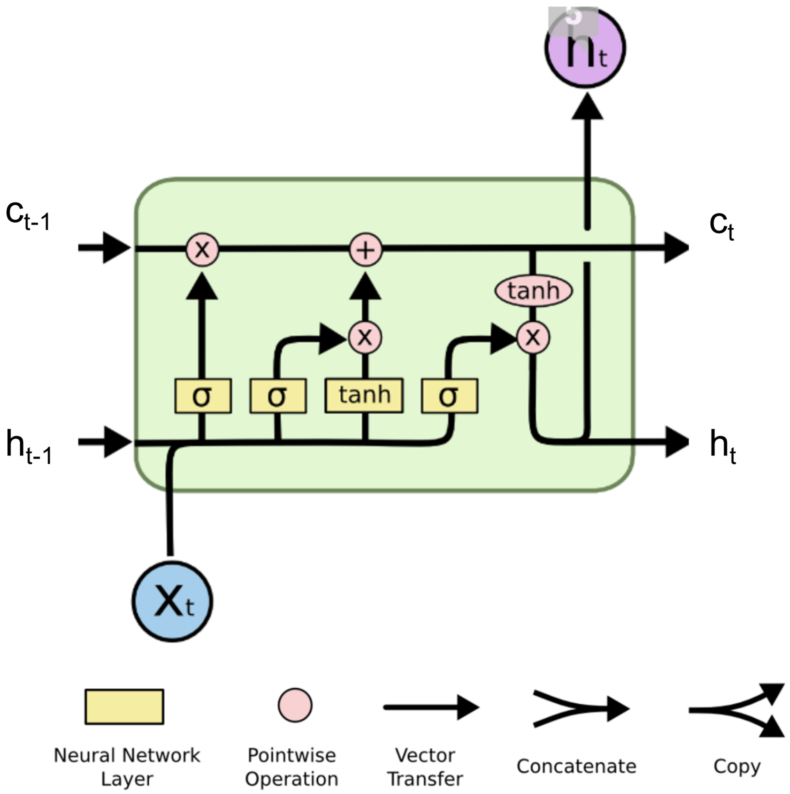
Long Short-Term Memory



$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right)$$
$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right)$$
$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)$$
$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right)$$
$$h_t = o_t \tanh(c_t)$$

Picture Credit: https://www.cs.toronto.edu/~graves/asru_2013.pdf

Long Short-Term Memory



$\mathcal{H}$

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right)$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right)$$

$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right)$$

$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right)$$

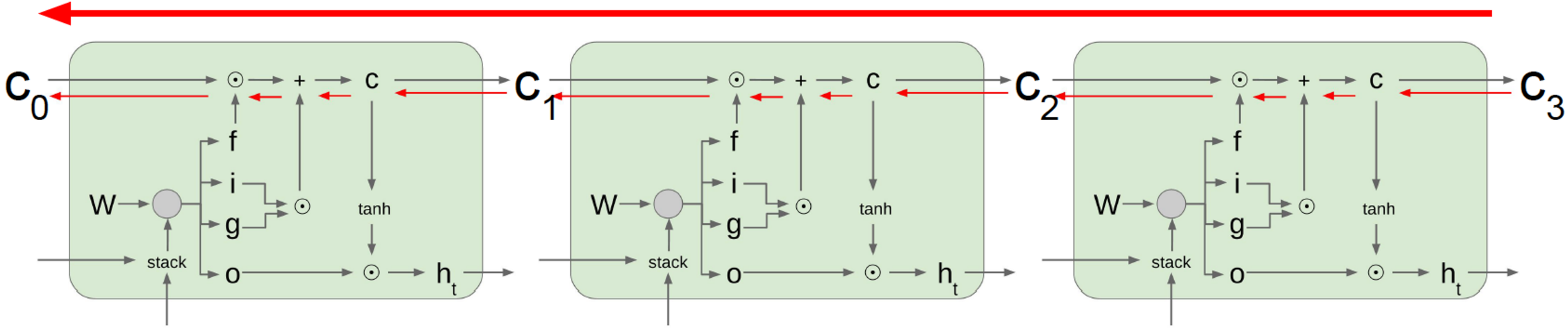
$$h_t = o_t \tanh(c_t)$$

Picture Credit: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

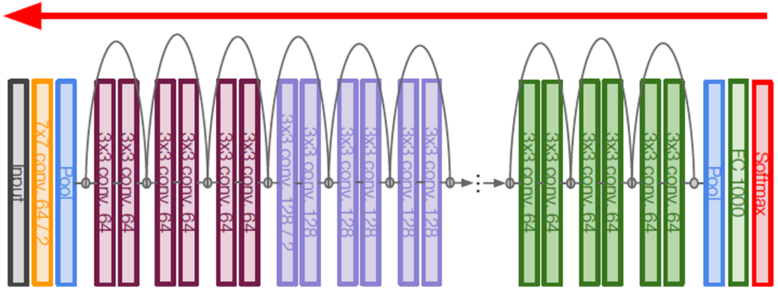
Long Short Term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Uninterrupted gradient flow!



Similar to ResNet!



In between:

Highway Networks

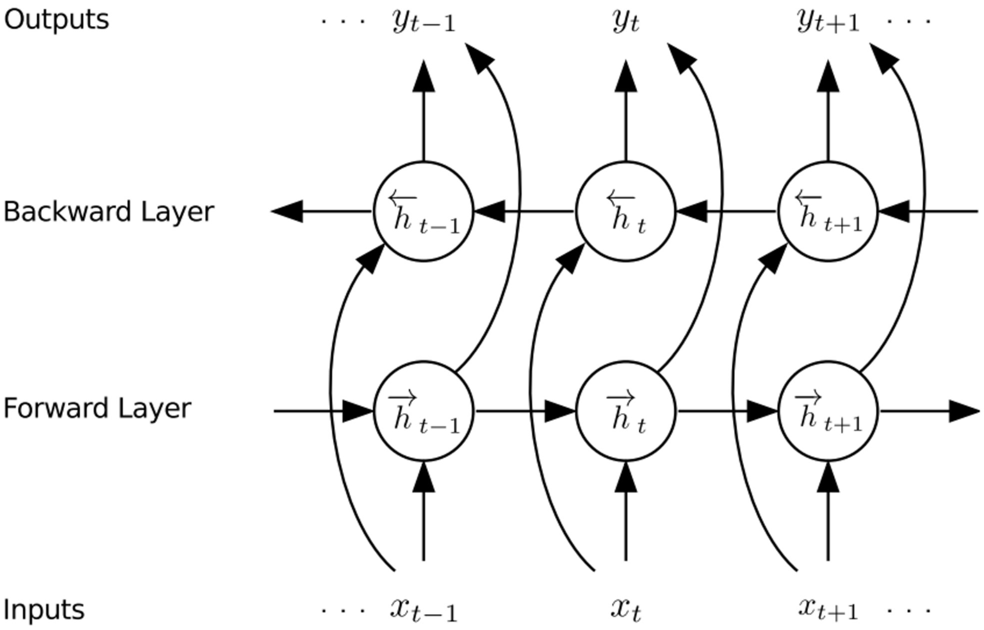
$$g = T(x, W_T)$$

$$y = g \odot H(x, W_H) + (1 - g) \odot x$$

Srivastava et al, "Highway Networks", ICML DL Workshop 2015

Slide Credit: Fei-Fei, Johnson, and Yeung, Stanford cs231n, 2019

Bidirectional LSTM



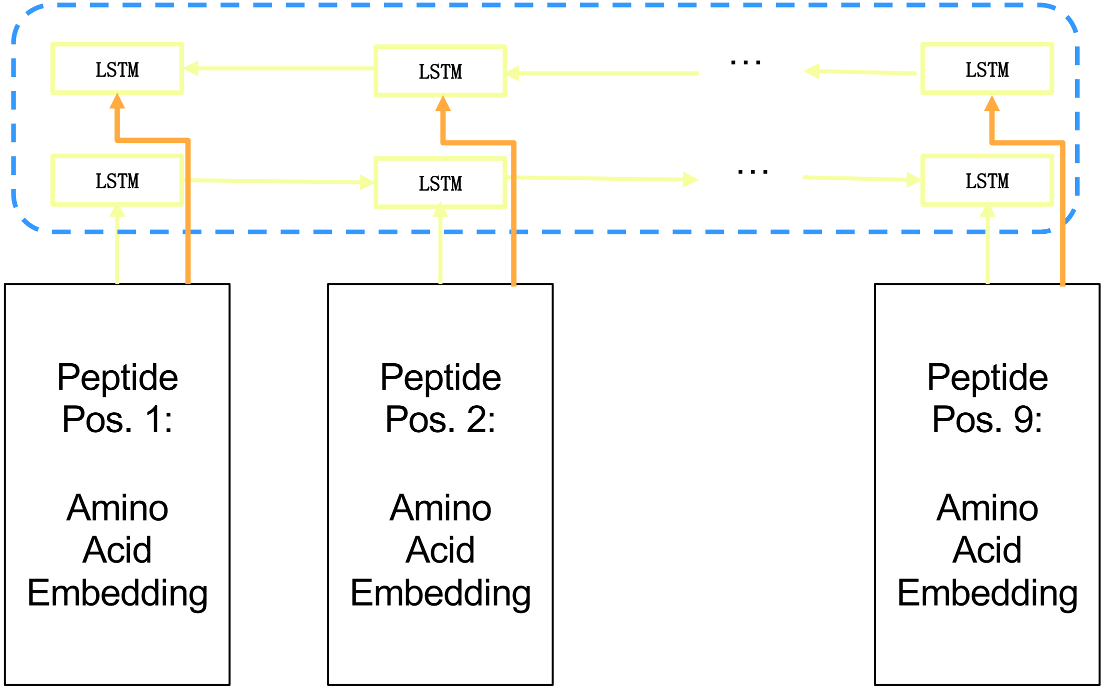
$$\overrightarrow{h}_t = \mathcal{H}\left(W_{x\overrightarrow{h}} x_t + W_{\overrightarrow{h}\overrightarrow{h}} \overrightarrow{h}_{t-1} + b_{\overrightarrow{h}}\right)$$

$$\overleftarrow{h}_t = \mathcal{H}\left(W_{x\overleftarrow{h}} x_t + W_{\overleftarrow{h}\overleftarrow{h}} \overleftarrow{h}_{t+1} + b_{\overleftarrow{h}}\right)$$

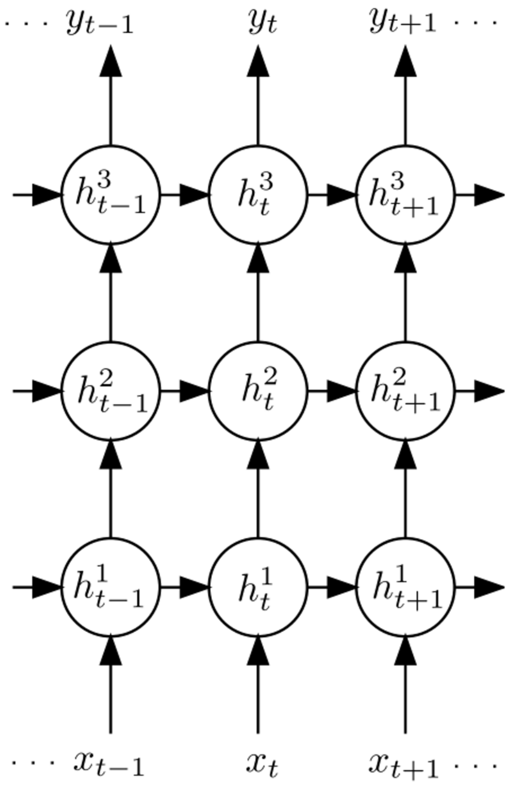
$$y_t = W_{\overrightarrow{h}y} \overrightarrow{h}_t + W_{\overleftarrow{h}y} \overleftarrow{h}_t + b_y$$

Picture Credit: https://www.cs.toronto.edu/~graves/asru_2013.pdf

Bidirectional LSTM



Deep LSTM



$$h_t^n = \mathcal{H}\left(W_{h^{n-1}h^n} h_t^{n-1} + W_{h^n h^n} h_{t-1}^n + b_h^n\right)$$

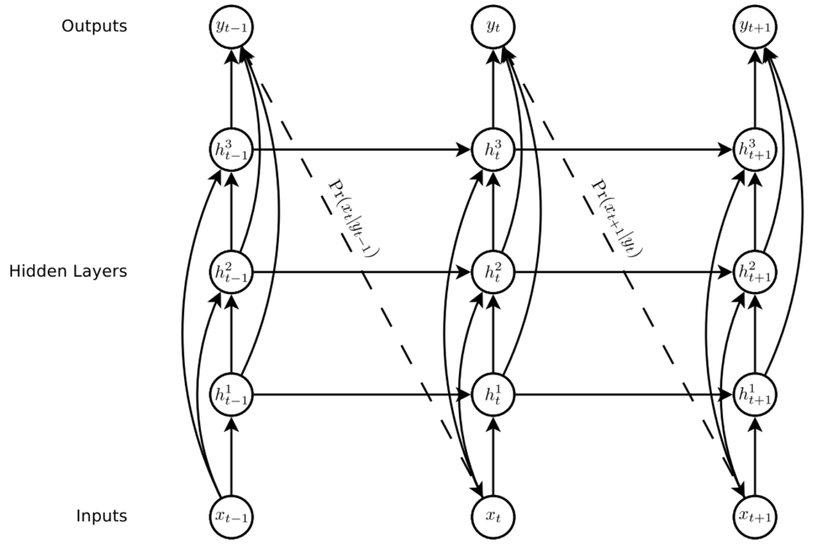
$$y_t = W_{h^N y} h_t^N + b_y$$

Picture Credit: https://www.cs.toronto.edu/~graves/asru_2013.pdf

Deep LSTM for Generating Complex Sequences

Generating text with characters or words as symbols

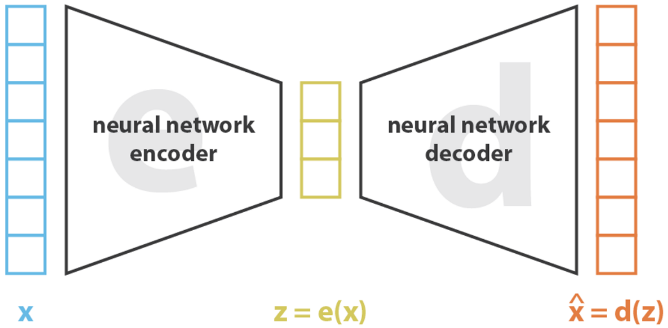
Generating handwriting with sequences of pen coordinates (x, y) and pen on/off whiteboard as input



Alex Graves, Generating Sequences With Recurrent Neural Networks. 2015

https://arxiv.org/pdf/1308.0850.pdf

Deep Encoder-Decoder Networks: Sequence-to-Sequence (Seq2Seq) Models



$$\text{loss} = \|x - \hat{x}\|^2 = \|x - d(z)\|^2 = \|x - d(e(x))\|^2$$

Illustration of an autoencoder with its loss function.

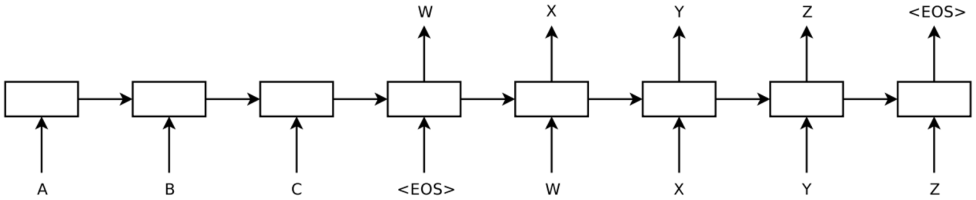


Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

Data Augmentation in Sequence-to-Sequence (Seq2Seq) Models for Machine Translation

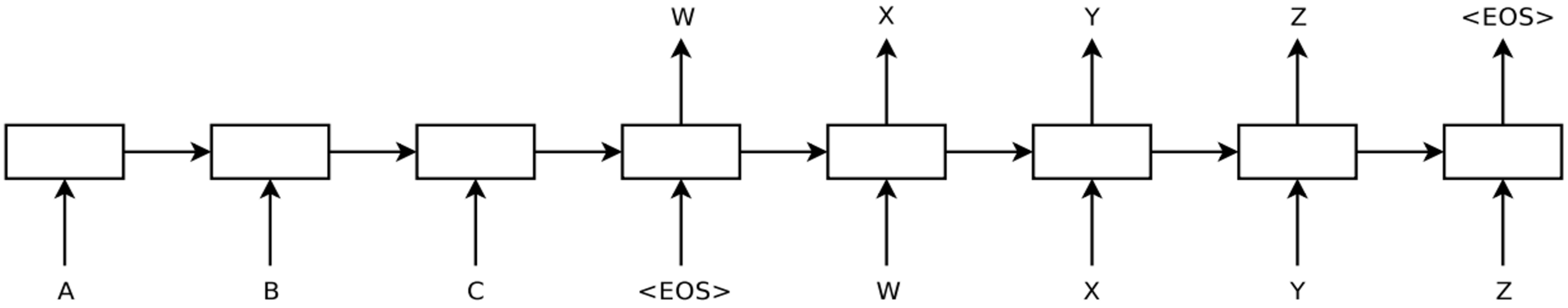


Figure 1: Our model reads an input sentence "ABC" and produces "WXYZ" as the output sentence. The model stops making predictions after outputting the end-of-sentence token. Note that the LSTM reads the input sentence in reverse, because doing so introduces many short term dependencies in the data that make the optimization problem much easier.

$$p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \ldots, y_{t-1})$$

representation v of the input sequence $(x_1, \ldots, x_T)$

$$\begin{aligned}
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \\
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \\
c_t &= f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \\
h_t &= o_t \tanh(c_t)
\end{aligned}$$

Summary of Topics Discussed

- Activation Functions
- Loss Functions
- Training deep feedforward neural networks with backpropagation and mini-batch SGD
- Convolution and pooling operations in CNN
- Network architectures such as AlexNet, VGG, ResNet
- Applications of supervised pre-trained CNNs
- Visualization of pre-trained CNN filters and receptive fields
- Recurrent Neural Networks, Sequence-to-Sequence Models
- Geoff Hinton, “Never stop coding.” Great discoveries are from practice.

The End

Next lecture:

Deep Learning III:

Deep Generative Models, VAE, and GAN