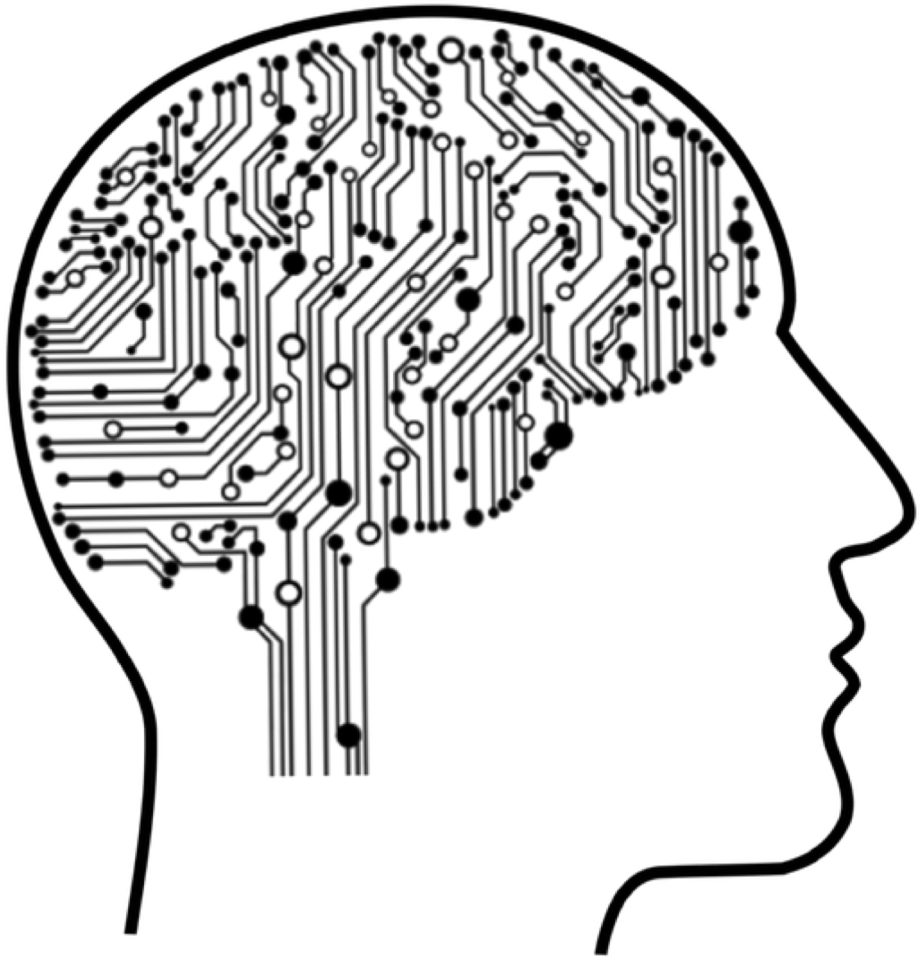
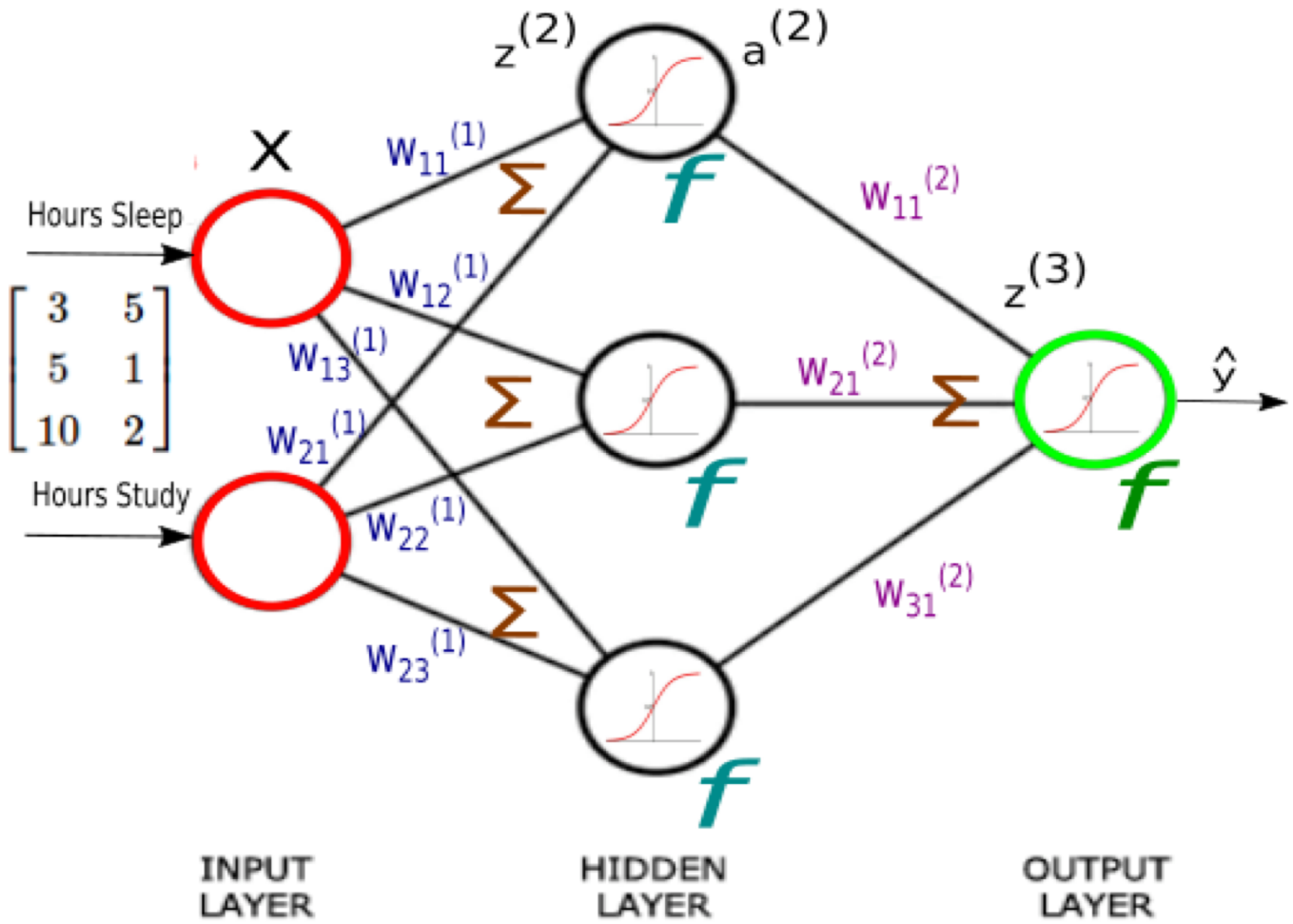
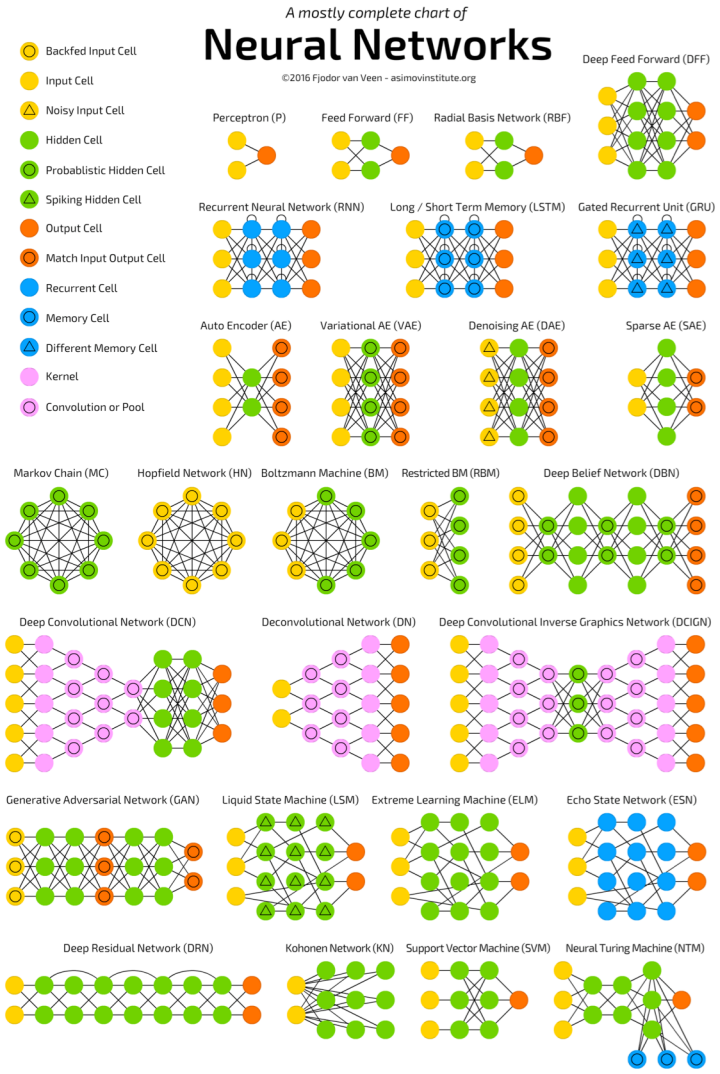
Neural Networks



Simple view neural network



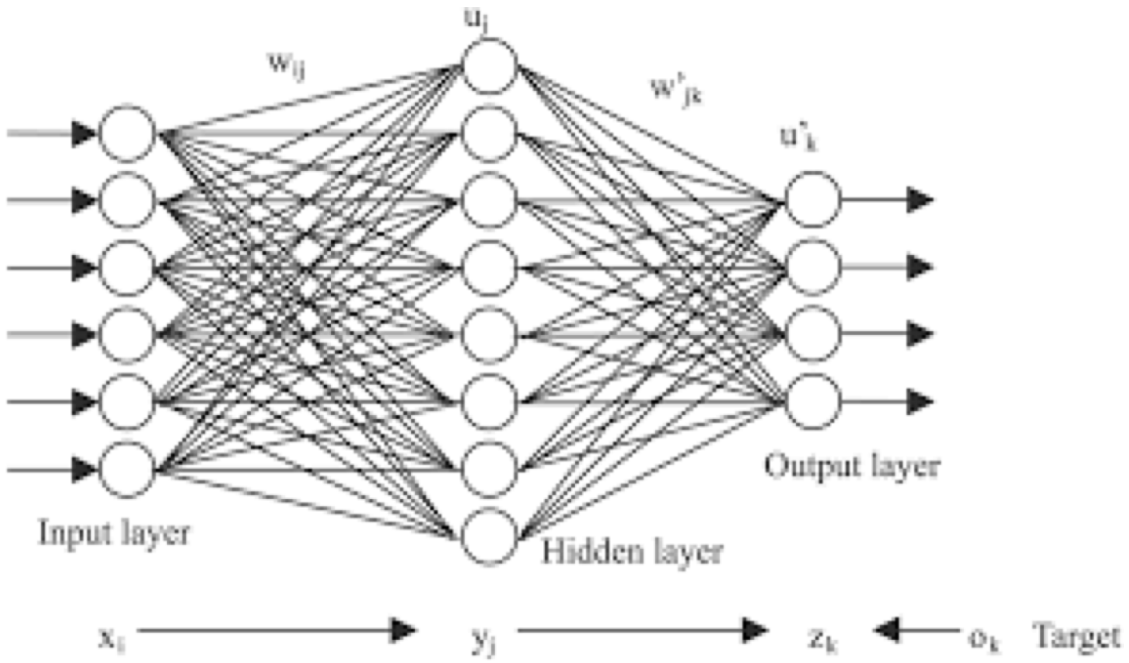


- Basic terms:
  - Deep Learning = Neural Networks
  - Deep Learning is a subset of Machine Learning
- Terms for neural networks:
  - MLP: Multilayer Perceptron
  - DNN: Deep neural networks
  - RNN: Recurrent neural networks
- LSTM: Long Short-Term Memory
- CNN or Conv Net: Convolutional neural networks
- DBN: Deep Belief Networks
- Neural network operations:
- Convolution
- Pooling
- Activation function
- Backpropagation

http://www.asimovinstitute.org/neural-network-zoo/

Artificial neural network

- ANN is inspired by the concept of biological nervous system.
- ANNs are the collection of computing elements (neurons) that may be connected in various ways.
- In ANNs the effect of the synapses is represented by the connection weight, which modulates the input signal.
- The architecture of artificial neural networks is a fully connected, three layered (input layer, hidden layer and output layer) structure of nodes in which information flows from the input layer to the output layer through the hidden layer.



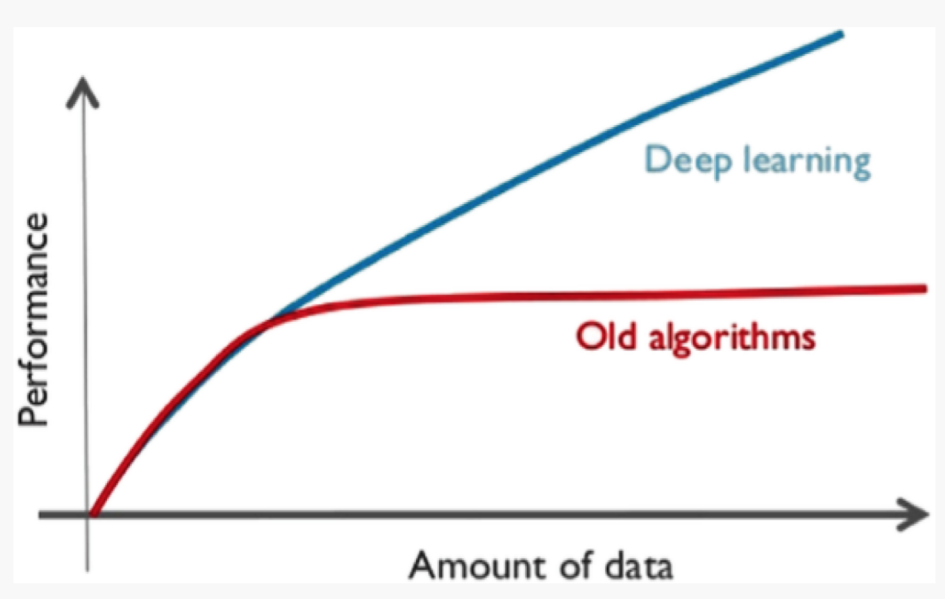
Artificial neural network

- ANNs are capable of linear and nonlinear classification.
- An ANN learns by adjusting the weights in accordance with the learning algorithms.
- It is capable to process and analyze large complex datasets, containing non-linear relationships.
- There are various types of artificial neural network architecture that are used in protein function prediction such as perceptron, multi-layer perceptron (MLP), radial basis function networks and kohonen self-organizing maps.

Analogy with keyboard

Deep Learning

- Deep learning network is essentially advanced form of ANN with multiple layers with sophisticated architecture.
- DL is the cornerstone technology behind products for images recognition, video annotation, voice recognition, personal assistant, automated translation and autonomous vehicle



Gradient descent to tune the weights

- While in some cases it is possible to find the global minimum of the cost function analytically (when it exists), in the great majority of cases we will have to use an optimization algorithm.
- Optimizers update the set of weights iteratively in a way that decreases the loss over time.
- The most commonly used approach is gradient descent, where we use the loss's gradient with respect to the set of weights

Update rule $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$

$\boldsymbol{\alpha}$ is learning rate, a positive number

Epoch - number of times iterate over the training dataset

An **epoch** is one complete presentation of the data to be learned to a learning machine. Let's make sense of what is happening during an **epoch.**

- Neural networks take the first image (or small group of images) and make a prediction about what it is (or they are).
- Their prediction is compared to the actual label of the image(s).
- The network uses information about the difference between their prediction and the actual label to adjust itself.
- The network then takes the next image (or group of images) and make another prediction.
- This new (hopefully closer) prediction is compared to the actual label of the image(s).
- The network uses information about the difference between this new prediction and the actual label to adjust again.
- This happens repeatedly until the network has looked at each image.

Epoch - comparison with human learning

Compare this to a human study session using flash cards:

- A student looks at a first flashcard and makes a guess about what is on the other side.
- They check the other side to see how close they were.
- They adjust their understanding based on this new information.
- The student then looks at the next card and makes another prediction.
- This new (hopefully closer) prediction is compared to the answer on the back of the card.
- The student uses information about the difference between this new prediction and the right answer to adjust again.
- This happens repeatedly until the student has tried each flashcard once.

Hidden layers and number of neurons

Number of hidden layers

- 0 - Only capable of representing linear separable functions or decisions.
- 1 - Can approximate any function that contains a continuous mapping from one finite space to another.
- 2 - Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.

Rule of thumb for the number of neurons in the hidden layers

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.
- $\frac{(m+n)}{2}$, m = output neurons, n = input neurons
- $\sqrt{m * n}$, m = output neurons, n = input neurons

Choice of activation function

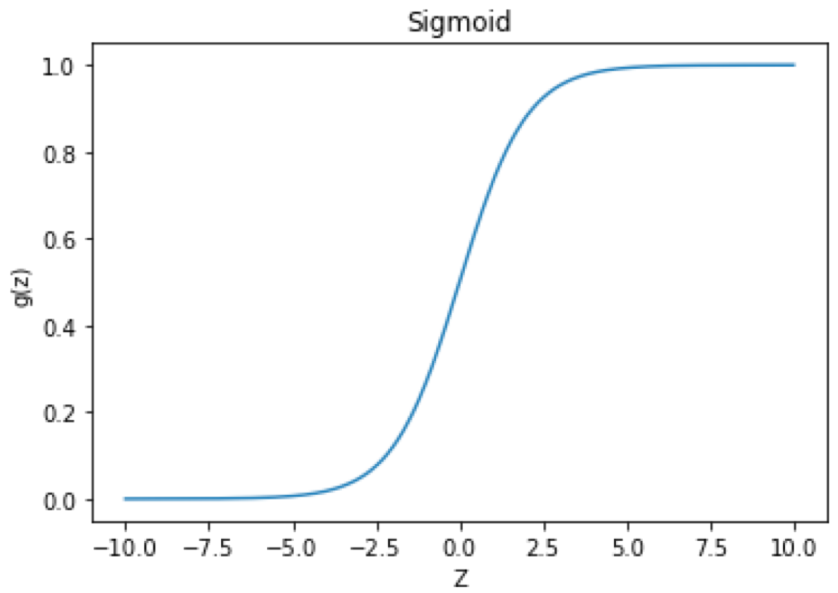
Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0. The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged. When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid.

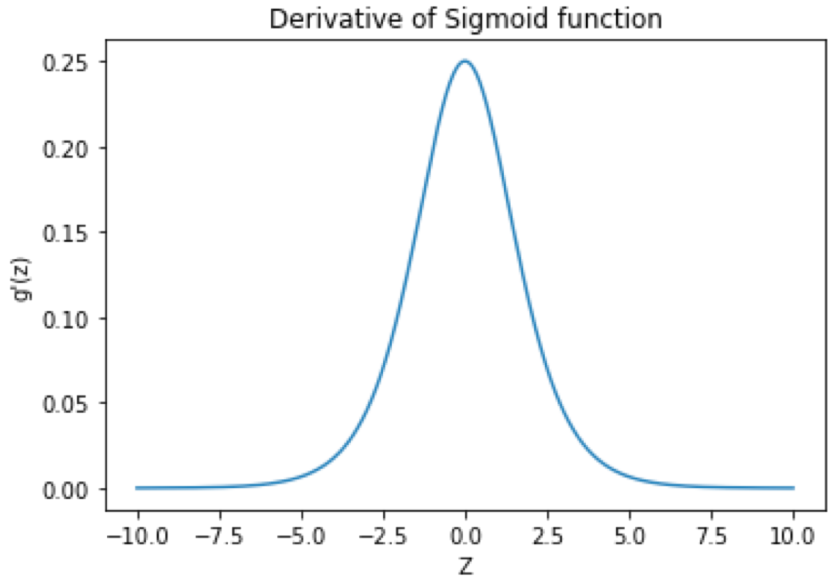
Sigmoid

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

Derivative

$$g'(z) = g(z)\big(1 - g(z)\big)$$
$$= a(1 - a)$$





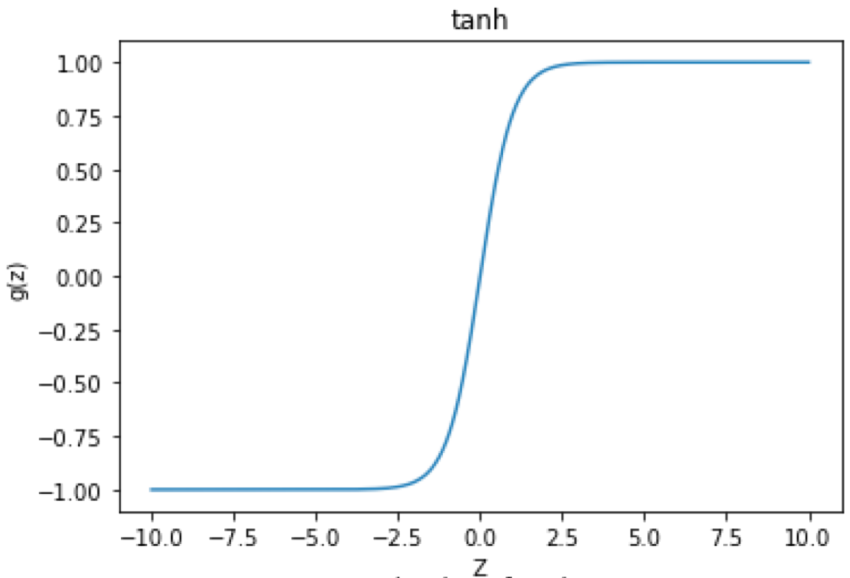
Tanh

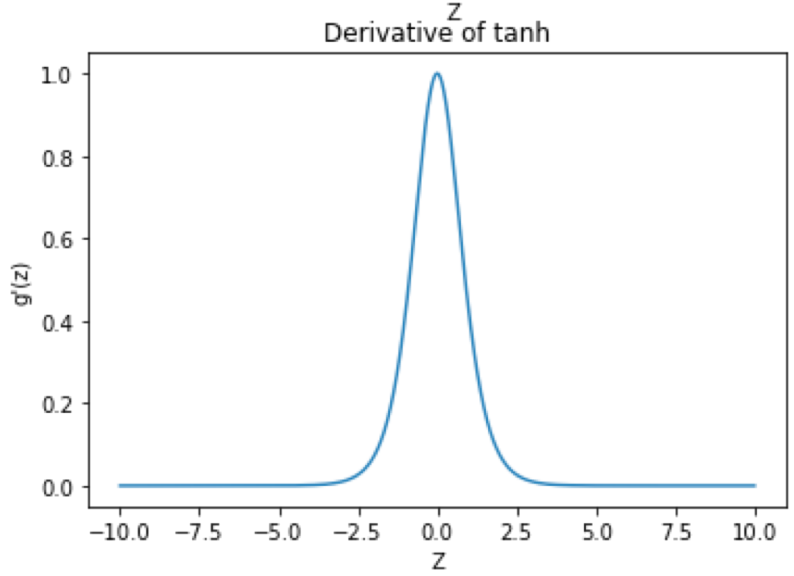
$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = a$$

Derivative

$$g'(z) = 1 - a^{2}$$

The tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.





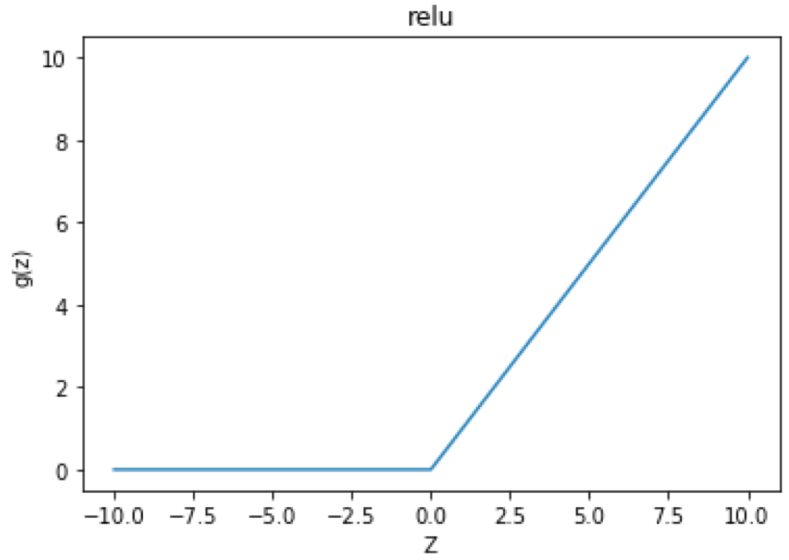
Relu - rectified linear unit, or ramp function

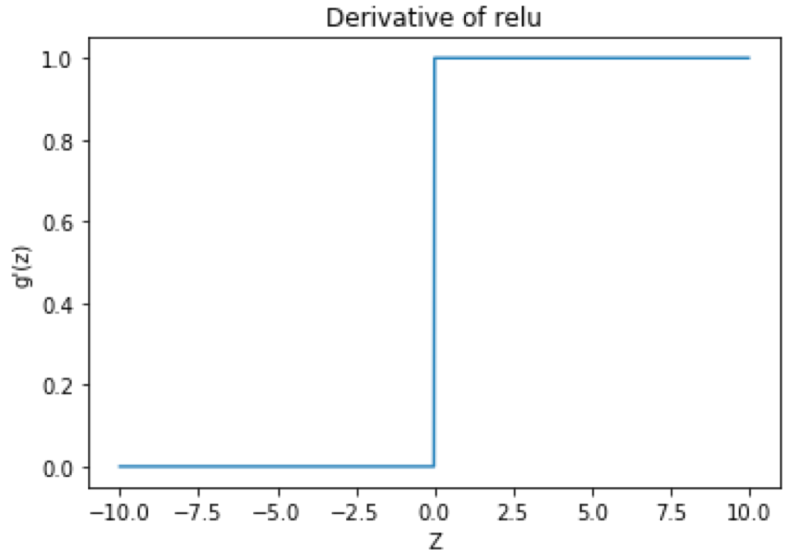
$$g(z) = \max(0, z)$$

Derivative

$$g'(z) = \begin{cases} -0, & z < 0 \\ 1, & z > 0 \\ undefined, & z = 0 \end{cases}$$

"Roses Are Red. Violets are Blue. Sigmoid is Good. But I do ReLU." - arXiv

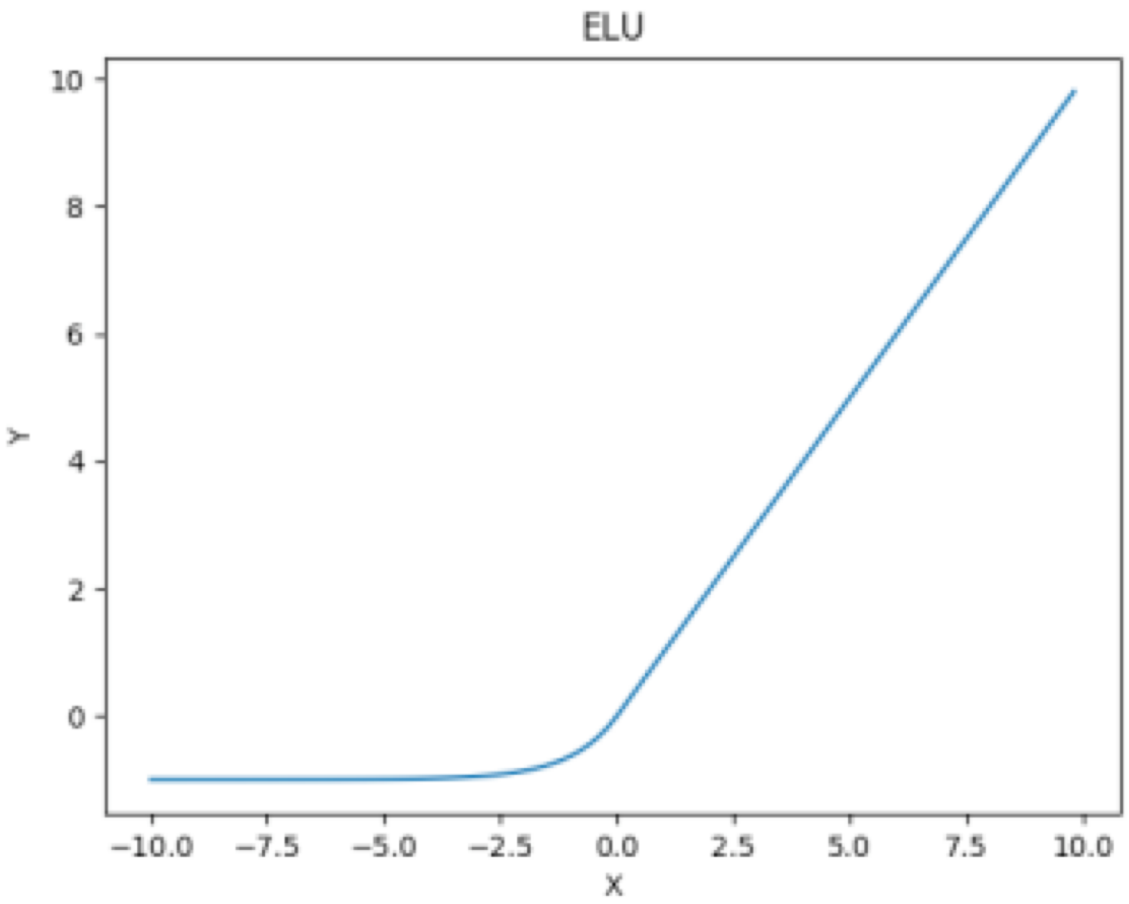




ELU - Exponential Linear Units

Problem with RELU is - it is not zero mean; it introduces a bias for the next layer which can slow down the learning

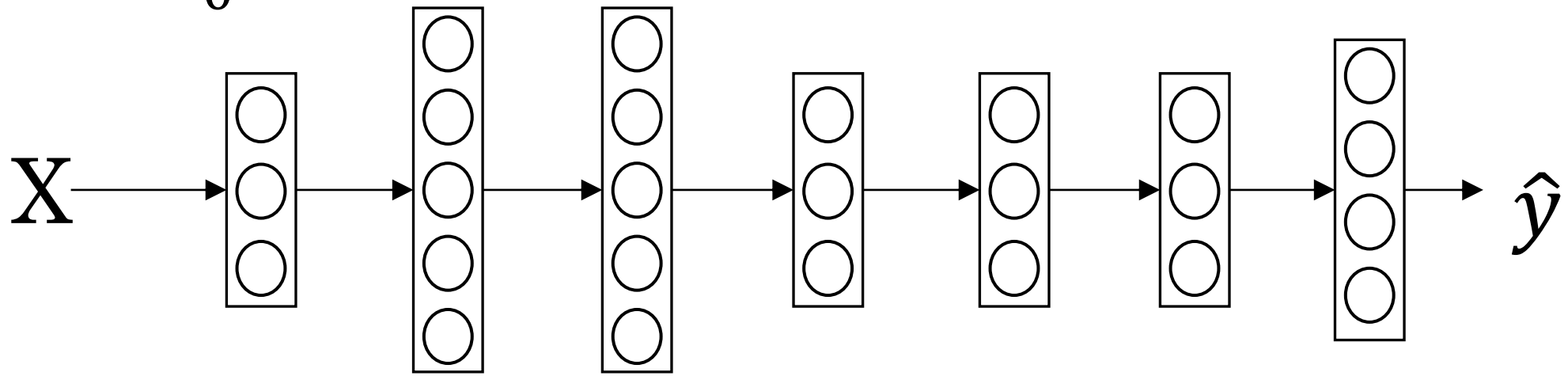
$$elu(x) = \begin{cases} x, if\ x \geq 0 \\ \boldsymbol{\alpha}(e^x - 1)\ otherwise \end{cases}$$



Softmax Classifier

Softmax classifier is used as activation function at the final layer for multi class classifications problems. It gives out a vector of probabilities for the classes.

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \hat{y}^{(i)} = \frac{e^{Z^{[l](i)}}}{\sum_{i=1}^{c} e^{Z^{[l](i)}}}$$, loss function = - $\sum_{i=1}^{n} y^{(i)} \log \hat{y}^{(i)}$



Entropy, Cross Entropy and KL Divergence

Entropy: $H_q = \sum_{i=n} p^{(i)} log\ p^{(i)}$

Cross Entropy $H_{p,q} = \sum_{i=n} p^{(i)} log q^{(i)}$

Divergence or relative cross entropy, $H_{p,q} = \sum_{i=n} p^{(i)} log\ \frac{q^{(i)}}{p^{(i)}}$

Deep Learning

- Deep learning is use a cascade of many layers of nonlinear processing units for feature extraction and transformation.
- Each successive layer uses the output from the previous layer as input.
- The algorithms may be supervised or unsupervised and applications include pattern analysis and classification.
- It is based on the learning of multiple levels of features or representations of the data.
- Higher level features are derived from lower level features to form a hierarchical representation.

Deep Learning

- It is a part of the broader machine learning field of learning representations of data.
- It learns multiple levels of representations that correspond to different levels of abstraction; the levels form a hierarchy of concepts.
- The composition of a layer of nonlinear processing units used in a deep learning algorithm depends on the problem to be solved.
- Layers that have been used in deep learning include hidden layers of an artificial neural network and sets of complicated propositional formulas.
- It may also include latent variables organized layer-wise in deep generative models such as the nodes in Deep Belief Networks and Deep Boltzmann Machines.

Deep Learning

- Deep learning algorithms are based on distributed representations.
- The underlying assumption behind distributed representations is that observed data are generated by the interactions of factors organized in layers.
- Deep learning adds the assumption that these layers of factors correspond to levels of abstraction or composition.
- Varying numbers of layers and layer sizes can be used to provide different amounts of abstraction.

Deep Learning

- Deep learning exploits this idea of hierarchical explanatory factors where higher level, more abstract concepts are learned from the lower level ones.
- These architectures are often constructed with a greedy layer-by-layer method.
- Deep learning helps to disentangle these abstractions and pick out which features are useful for learning.
- For supervised learning tasks, deep learning methods obviate feature engineering, by translating the data into compact intermediate representations akin to principal components, and derive layered structures which remove redundancy in representation.

Deep Learning

- Many deep learning algorithms are applied to unsupervised learning tasks.
- This is an important benefit because unlabeled data are usually more abundant than labeled data.
- Examples of deep structures that can be trained in an unsupervised manner are neural history compressors and deep belief networks.

Deep learning framework

Name	Origin	Language Interface	GPU Support	Description
Caffe	Berkley	C++, Python	Yes	
CNTK	Microsoft	C++, .Net	Yes	
DL4J	Skymind	C++, Java	Yes	
Keras		Python, R	Yes	
Theano	U Montreal	Python	Yes	
MXNet	Apache Software		Yes	
PaddlePaddle	Baidu		Yes	
TensorFlow	Google	Python, C, Java, JS	Yes	
Torch		C, Lua		
PyTorch	Facebook	Python	Yes	

Factor to choose the right framework

- Choosing deep learning frameworks
- Ease of programming (development and deployment)
- Running speed
- Truly open (open source with good governance)
- Language of preference
- Field of application - vision, speech, advertisement etc.

Initialize weights

- Logistic Regression doesn't have a hidden layer. If you initialize the weights to zeros, the first example x fed in the logistic regression will output zero but the derivatives of the Logistic Regression depend on the input x (because there's no hidden layer) which is not zero. So at the second iteration, the weights values follow x's distribution and are different from each other if x is not a constant vector.
- Neural network has hidden layers. If you decide to initialize the weights and biases to be zero, each neuron in the first hidden layer will perform the same computation. So even after multiple iterations of gradient descent each neuron in the layer will be computing the same thing as other neurons.
- If you initialize the weights to relative large values. This will cause the inputs of the tanh to also be very large, thus causing gradients to be close to zero. The optimization algorithm will thus become slow.

Exploding and Vanishing Weights

- If the weights in a network start too small, then the signal shrinks as it passes through each layer until it’s too tiny to be useful.
- If the weights in a network start too large, then the signal grows as it passes through each layer until it’s too massive to be useful.
- It turns out that initialization is surprisingly important. A marked difference can appear with only 3-4 layers in the network.

Xavier and He initialization of weights

- By using Xavier initialization, we make sure that the weights are not too small but not too big to propagate accurately the signals.

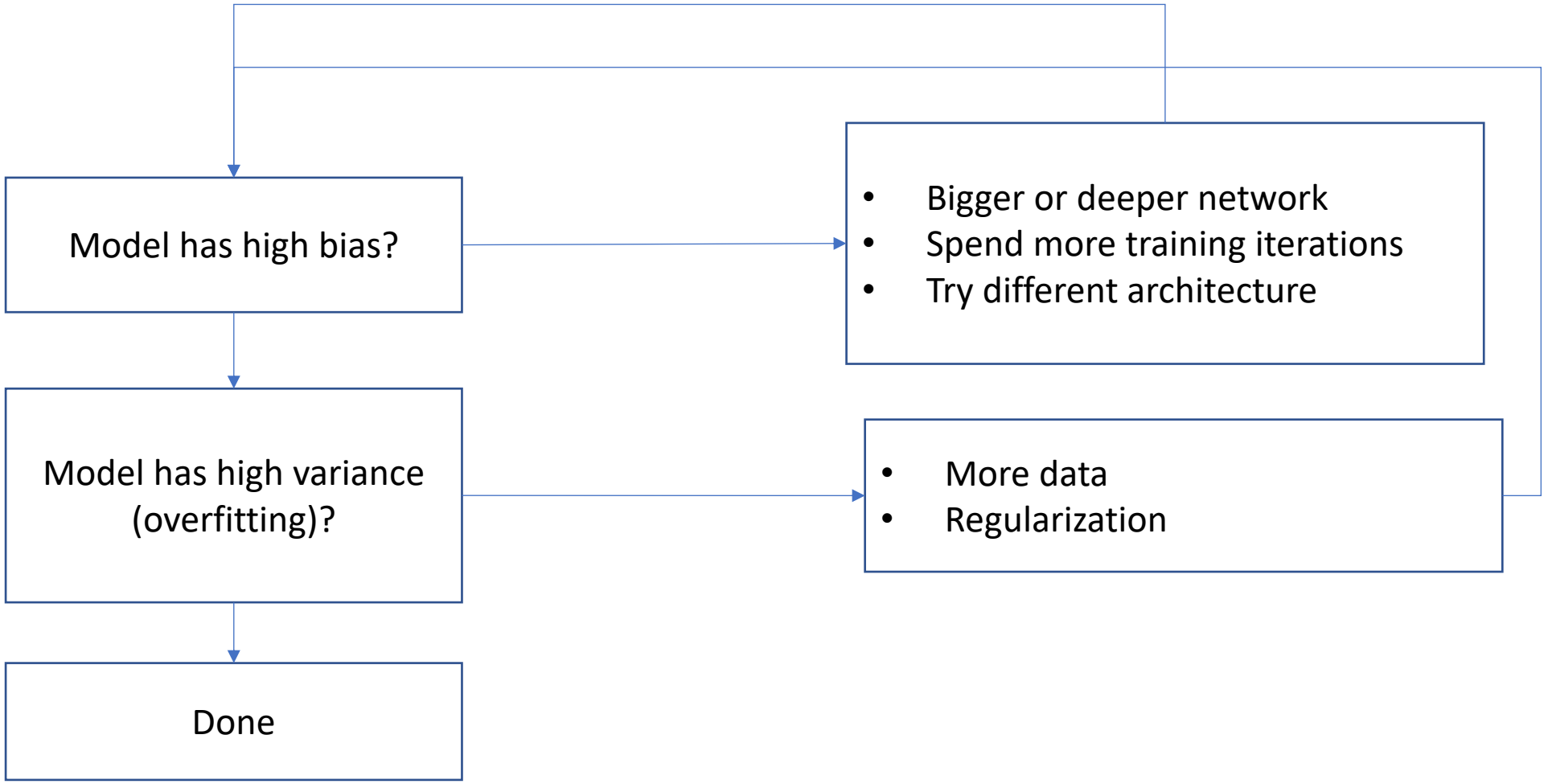
$$var\left(w^{[l]}\right) = \frac{2}{n^{[l-1]} + n^{[l]}}$$

- He initialization (better choice for ReLU or tanh activation function)

$$var\left(w^{[l]}\right) = \frac{2}{n^{[l-1]}}$$

Regularize Deep Neural Net

Bias Variance Trade Off in Deep Learning



Regularization

Cross entropy $cost, J = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right)$

$\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) = -y^{(i)}\, p(y = c)$

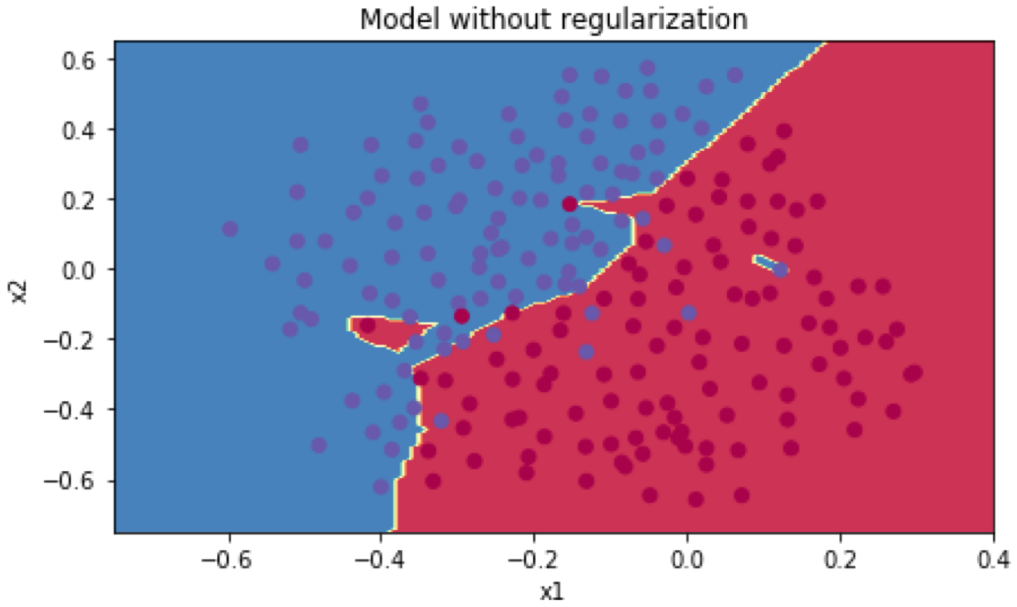
$$p(y = c) = \frac{e^{z_c^{(i)}}}{\sum_{c=1}^{C} e^{z_c^{(i)}}}$$

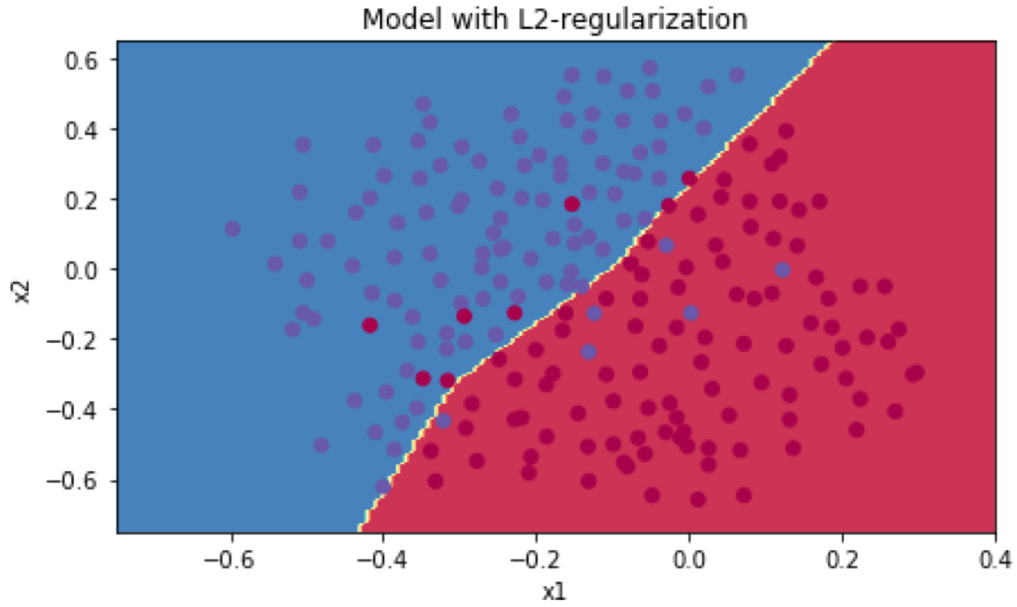
L2 Regularized Cost Function

$$J = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}\left(y^{(i)}, \hat{y}^{(i)}\right) + \frac{\lambda}{2m}\|W\|_F^2$$

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

Effect of Regularization

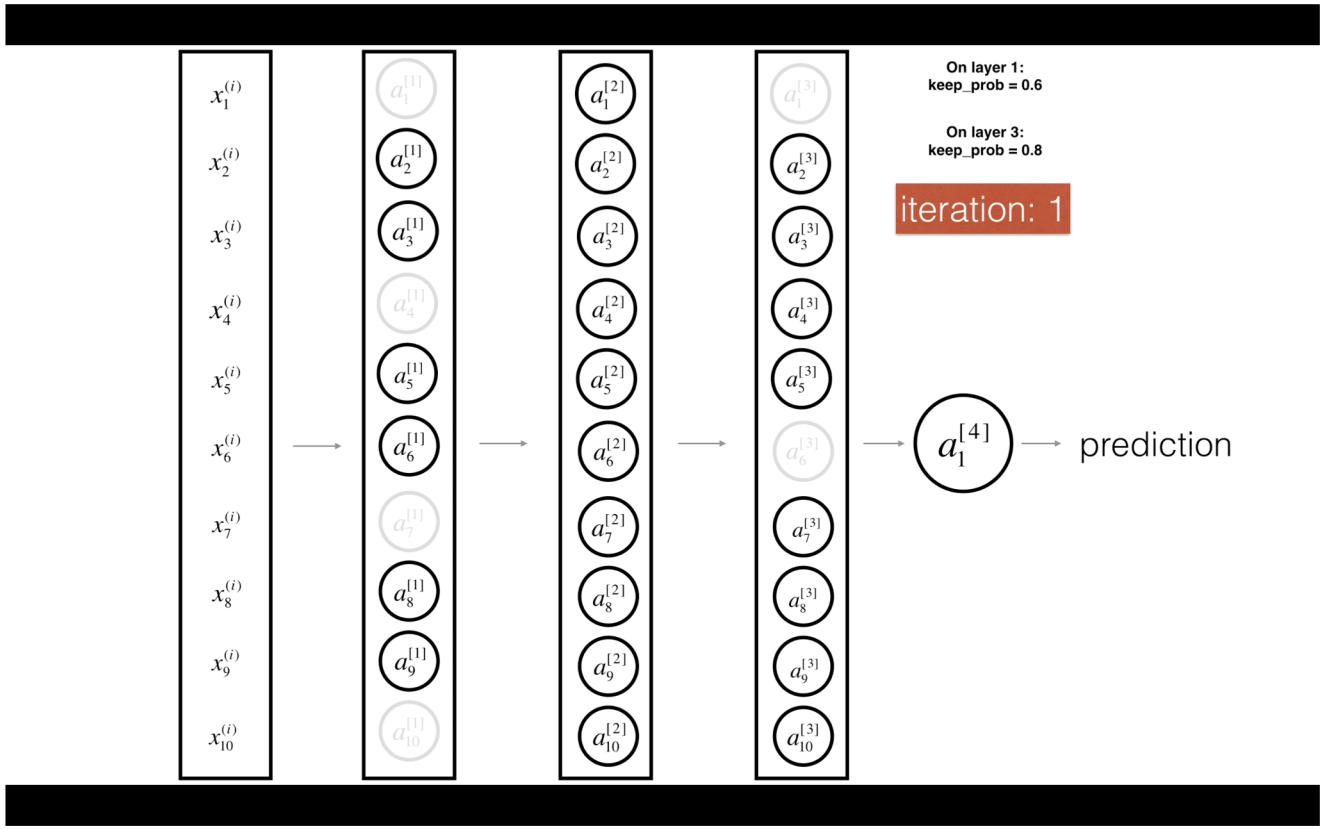




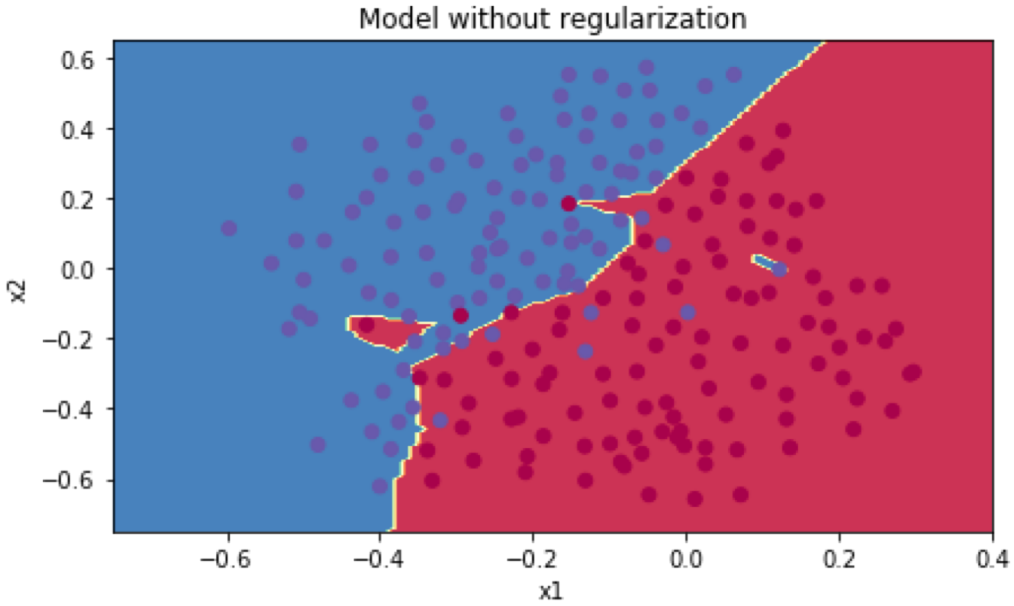
L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

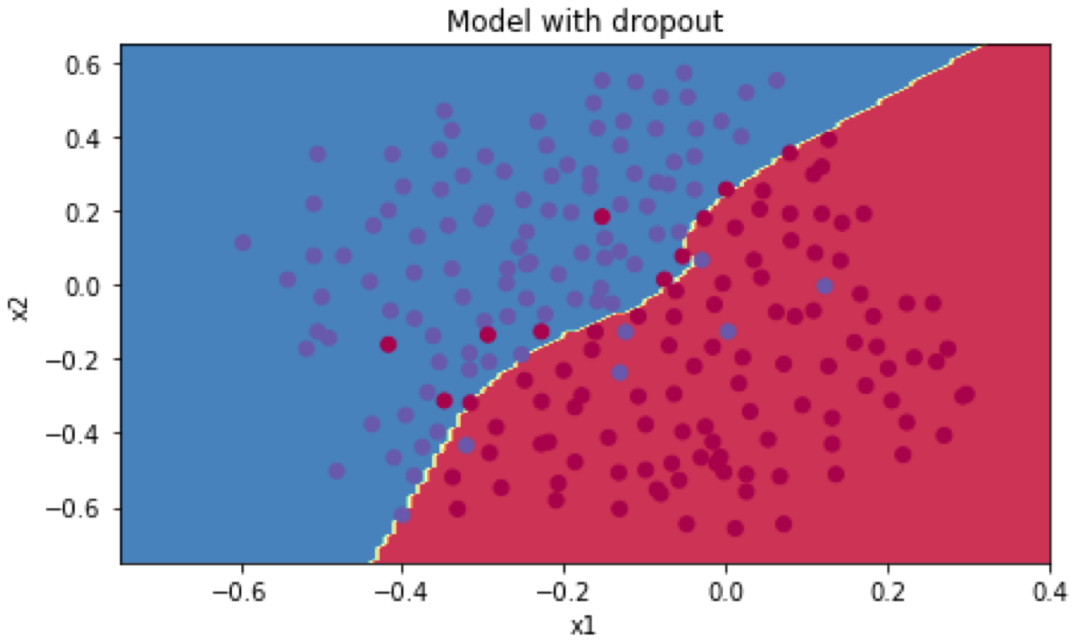
Dropout

- Dropout is a widely used regularization technique that is specific to deep learning.
- It randomly shuts down some neurons in each iteration.
- The dropped neurons don't contribute to the training in both the forward and backward propagations of the iteration.
- The idea behind drop-out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.



Effect of Dropout



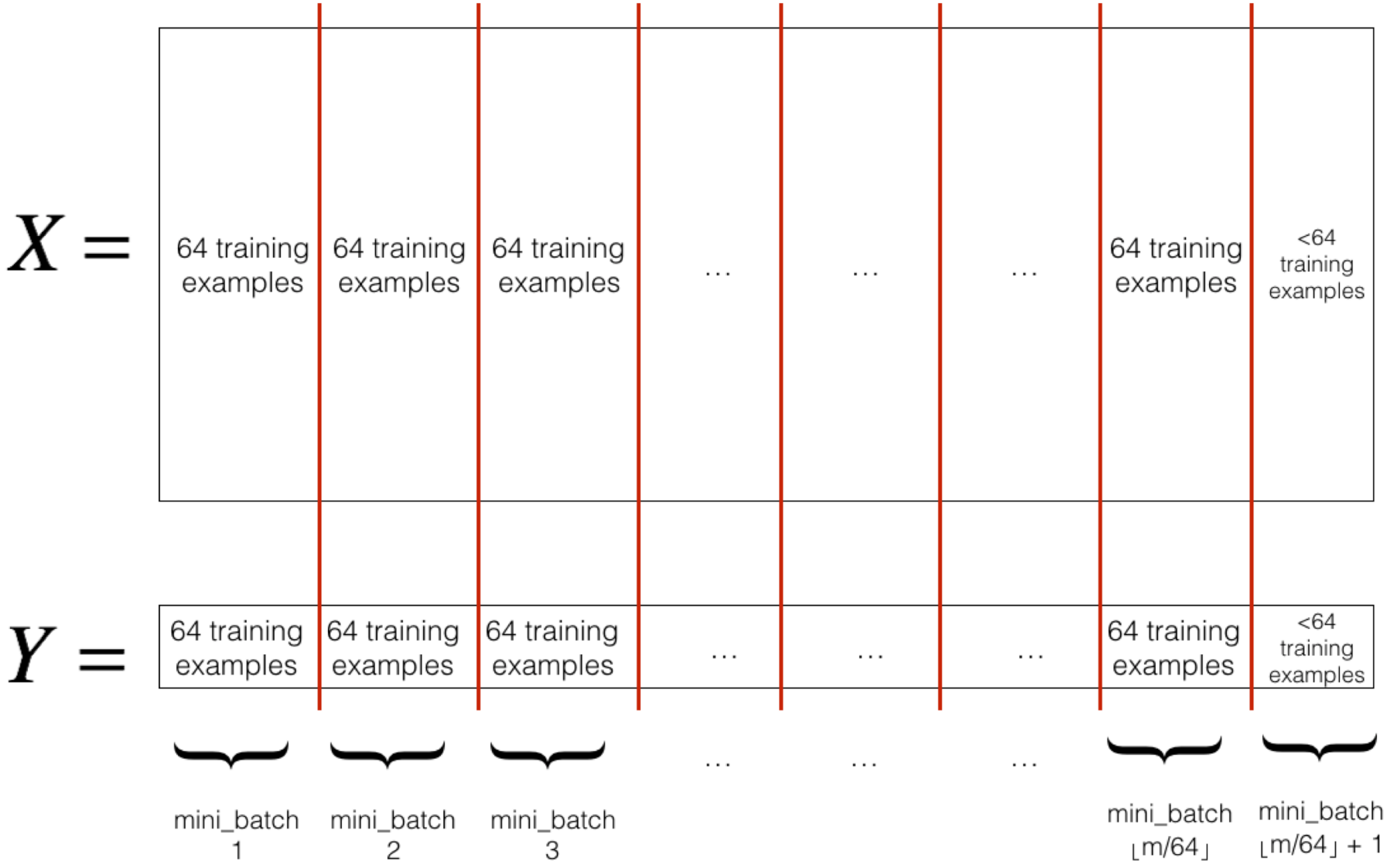


Regularization Summary Table

Model	Train Accuracy	Test Accuracy
3-layer NN without regularization	95%	91.5%
3-layer NN with L2-regularization	94%	93%
3-layer NN with dropout	93%	95%

Note that regularization hurts training set performance! This is because it limits the ability of the network to overfit to the training set. But since it ultimately gives better test accuracy, it is helping your system.

Mini Batch

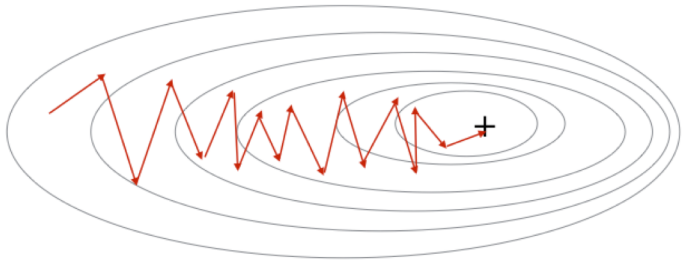

$X =$
64 training examples
64 training examples
64 training examples
…
…
…
64 training examples
<64 training examples
$Y =$
64 training examples
64 training examples
64 training examples
…
…
…
64 training examples
<64 training examples
mini_batch 1
mini_batch 2
mini_batch 3
mini_batch ⌊m/64⌋
mini_batch ⌊m/64⌋ + 1

Mini Batch, Stochastic and Batch

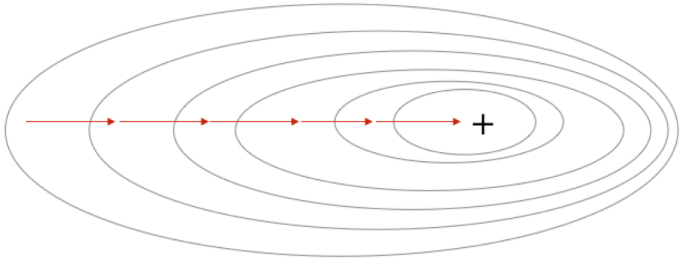
Batch Gradient Descent	Mini Batch	Stochastic
• At each iteration, calculate $\nabla\theta$ over entire training dataset and update $\theta:=\theta - \eta\nabla$ • Takes longer for iteration • Data maybe too much to to fit in memory • Suitable for training set < 2000 records	• At each iteration, calculate $\nabla\theta$ over a small batch of training examples (subset from the entire training set) and update $\theta:=\theta - \eta\nabla$ • Batch size is generally between 64-512 (2^n) • Oscillates near the global minimum • Takes advantage of Vectorized operations • Suitable for big dataset	• It is a mini batch with batch size = 1 • Lose the speed of vectorized operation • Converges faster to minimum faster since the θ are update more often • Oscillates around the global minimum

Stochastic vs Mini Batch vs Batch

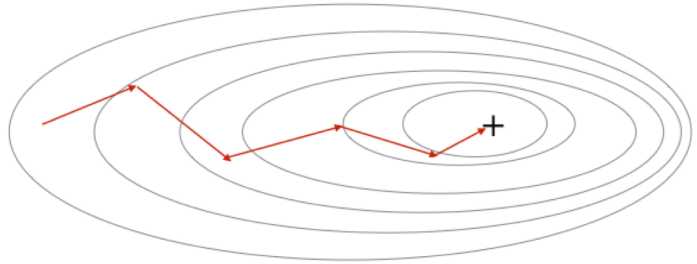
Stochastic Gradient Descent



Gradient Descent

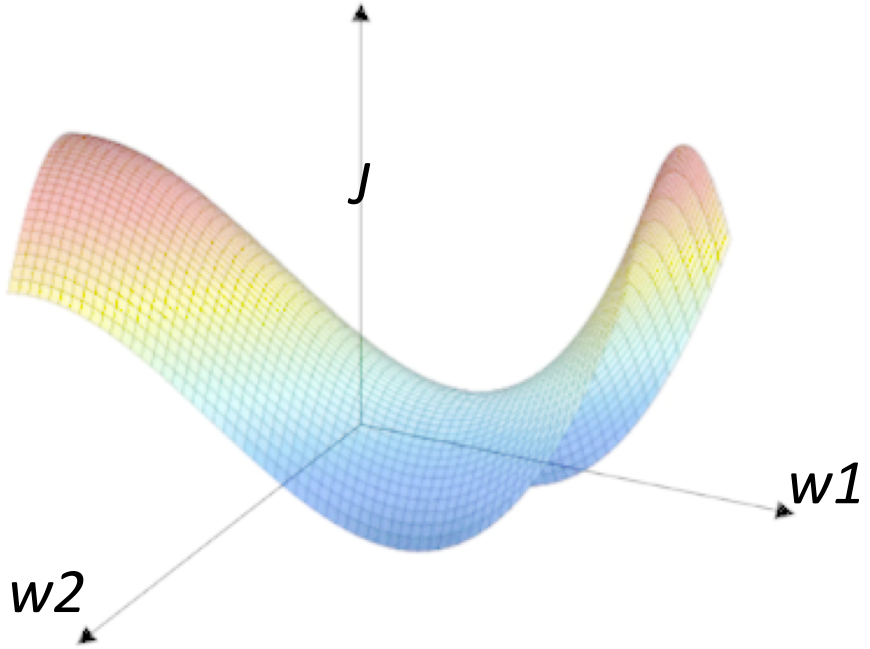


Mini-Batch Gradient Descent

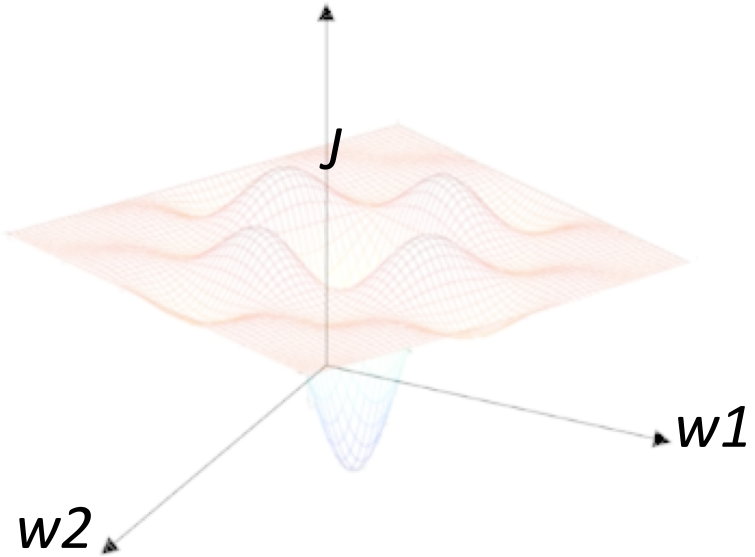


Gradient descent optimizers

Problem of local optima and saddle points



Saddle Point: cost functions does not change over long period of time. It is a common problem in deep learning. Momentum methods help solve such problems



Local optima: less common for high dimensional data. One common trick is to shuffle the full training data to avoid such problems.

Gradient descent with momentum

$$W = W - \alpha v_{dW}$$
$$v_{dW} = \beta\, v_{dW} + (1-\beta) dW$$

α, β are hyper parameters, α is learning rate, β is usually 0.9

RMS Prop

$$S_{dW} = \beta S_{dW} + (1-\beta)(dW)^2$$

$$W = W - \alpha \frac{dW}{\sqrt{S_{dW} + \varepsilon}}$$

Hyper parameters: α, β
β is usually 0.9, α is learning rate
Epsilon ε is added to avoid divide by zero and usually 10^{-8}

ADAM (Adaptive Moment Estimation)

- Adam combines the advantages of RMSProp and momentum and used for mini batch gradient descent
- The learning rate parameter is usually tuned
- Usually use default values for hyper parameters β_1, β_2 and ε ($\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-8}$)

$$v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1)\frac{dJ}{dW^{[l]}}$$

$$v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1 - \beta_1^2}$$

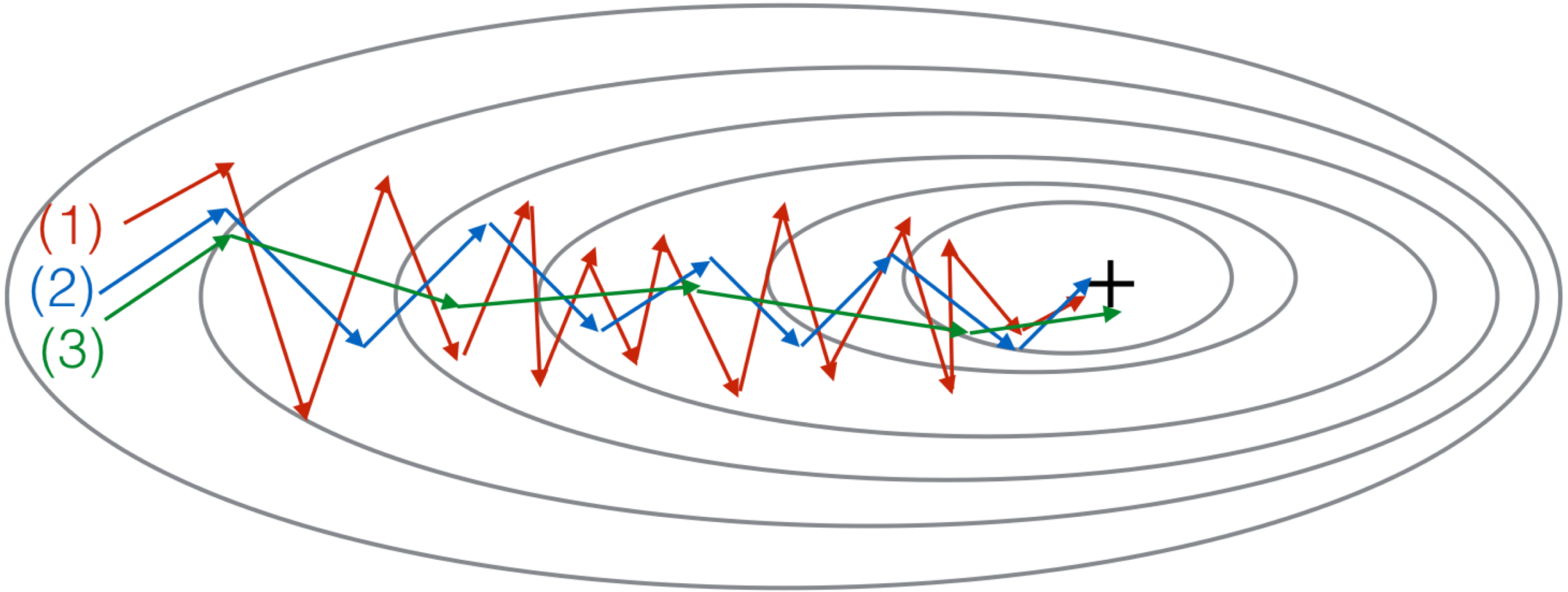
$$S_{dW^{[l]}} = \beta_2 S_{dW^{[l]}} + (1 - \beta_2)(\frac{dJ}{dW^{[l]}})^2$$

$$S_{dW^{[l]}}^{corrected} = \frac{S_{dW^{[l]}}}{1 - \beta_2^2}$$

$$W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{S_{dW^{[l]}}^{corrected}} + \varepsilon}$$

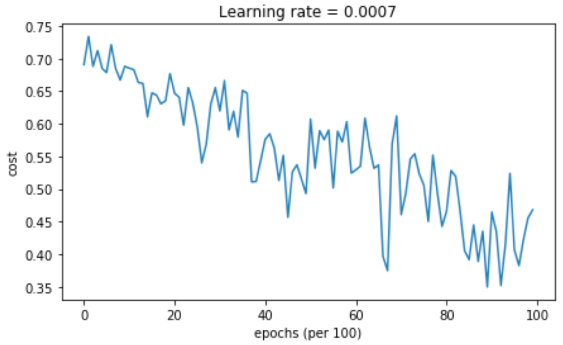
Adam paper: https://arxiv.org/pdf/1412.6980.pdf

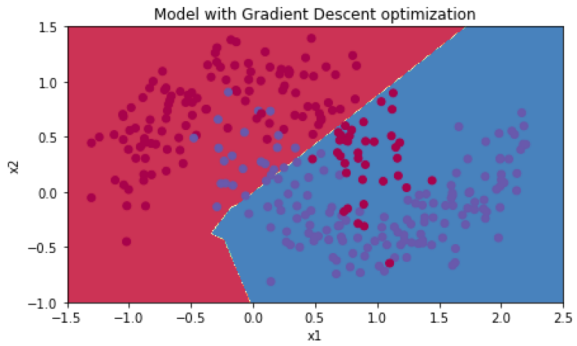
Gradient Descent Optimizer



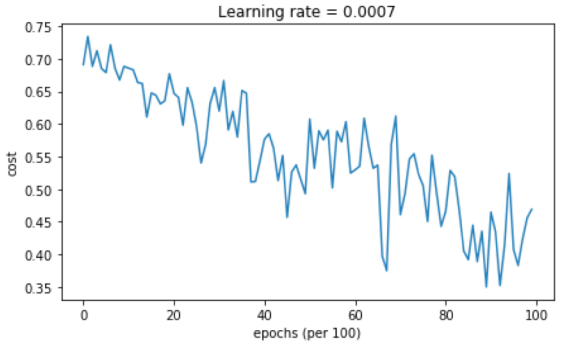
(1) gradient descent with momentum (small β)
(2) gradient descent with momentum (small β),
(3) gradient descent

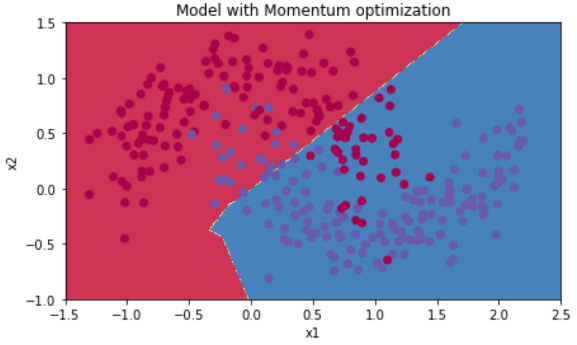
Mini Batch, With Momentum and Adam



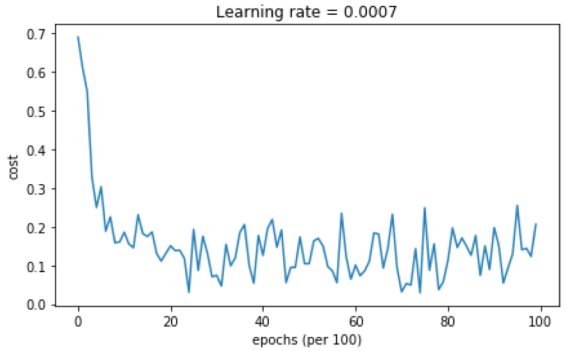


Gradient Descent with Mini Batch





Gradient Descent with Momentum Optimizer



Gradient Descent with Adam Optimizer

Accuracy Scores

Mini - 79.7%
Momentum - 79.7%
Adam - 94%

Learning Rate Decay

$$\alpha = \frac{\alpha_0}{1 + decay\ rate\ * t}$$

This is known inverse time scale decay

$$\alpha = \frac{1}{\sqrt{t}}\alpha_0$$

$$\alpha = 0.95^t \alpha_0$$

This is known exponential decay

t is the epoch number, epoch is one pass over entire training set

What if the cost functions remains flat

- Try better random initialization for the weight (He method)
- Try tuning the learning rate
- Try using Adam
- Try mini batch gradient descent

Batch Normalization

Normalization: local response normalization across channels (LRN), batch normalization.

Batch normalization helps in two ways

- For very deep network, the batch normalization helps converge faster
- There is a slight side-effect of regularization.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

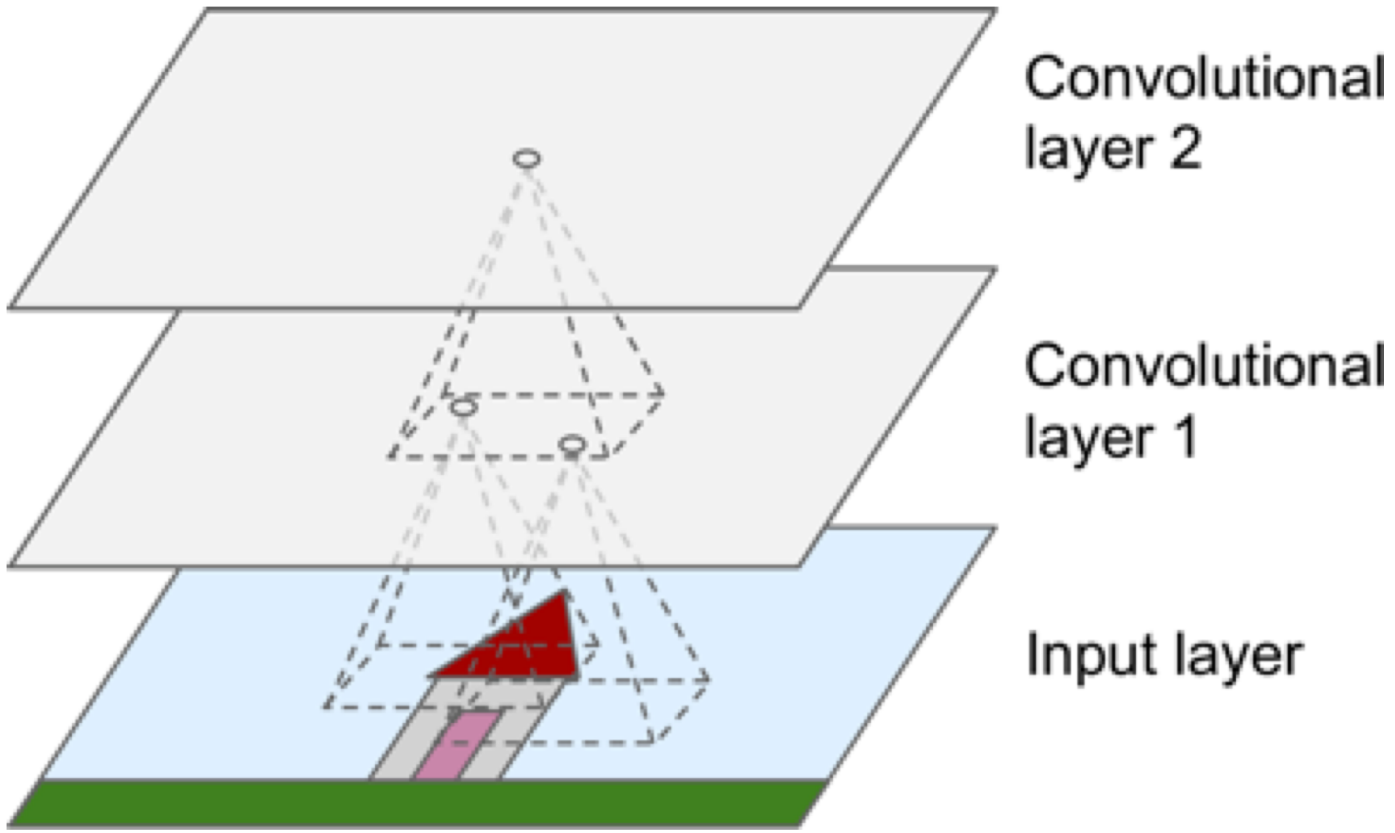
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

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

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

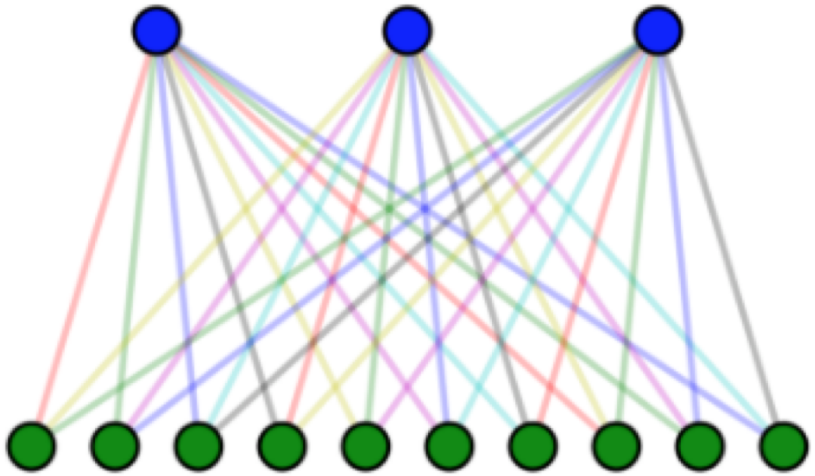
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Convolutional Neural Network



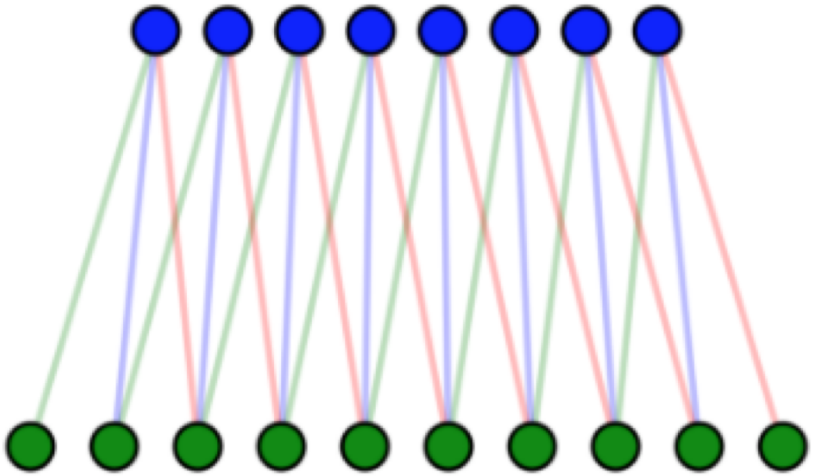
Fully Connected vs CNN

Fully Connected



Each unit is connected to all of the units in the previous layer

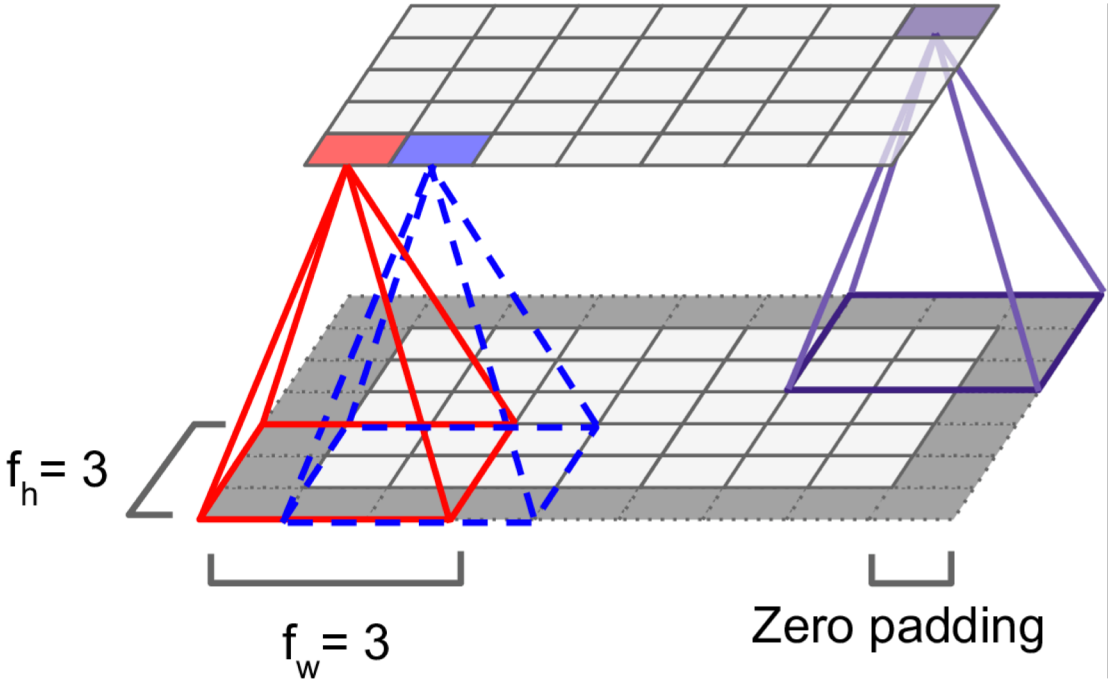
Convolutional Layer



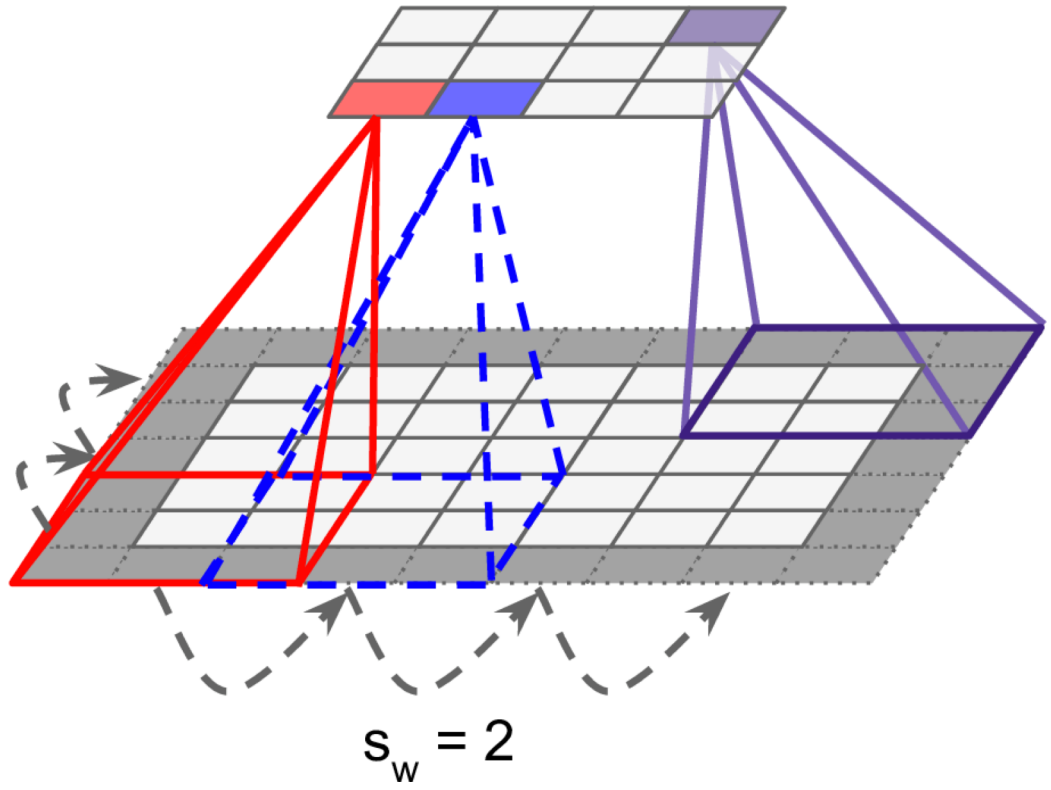
Each unit is connected to a (typically small) number of nearby units in the previous layer. Furthermore, all units are connected to the previous layer in the same way, with the exact same weights and structure.

Convolutional layer

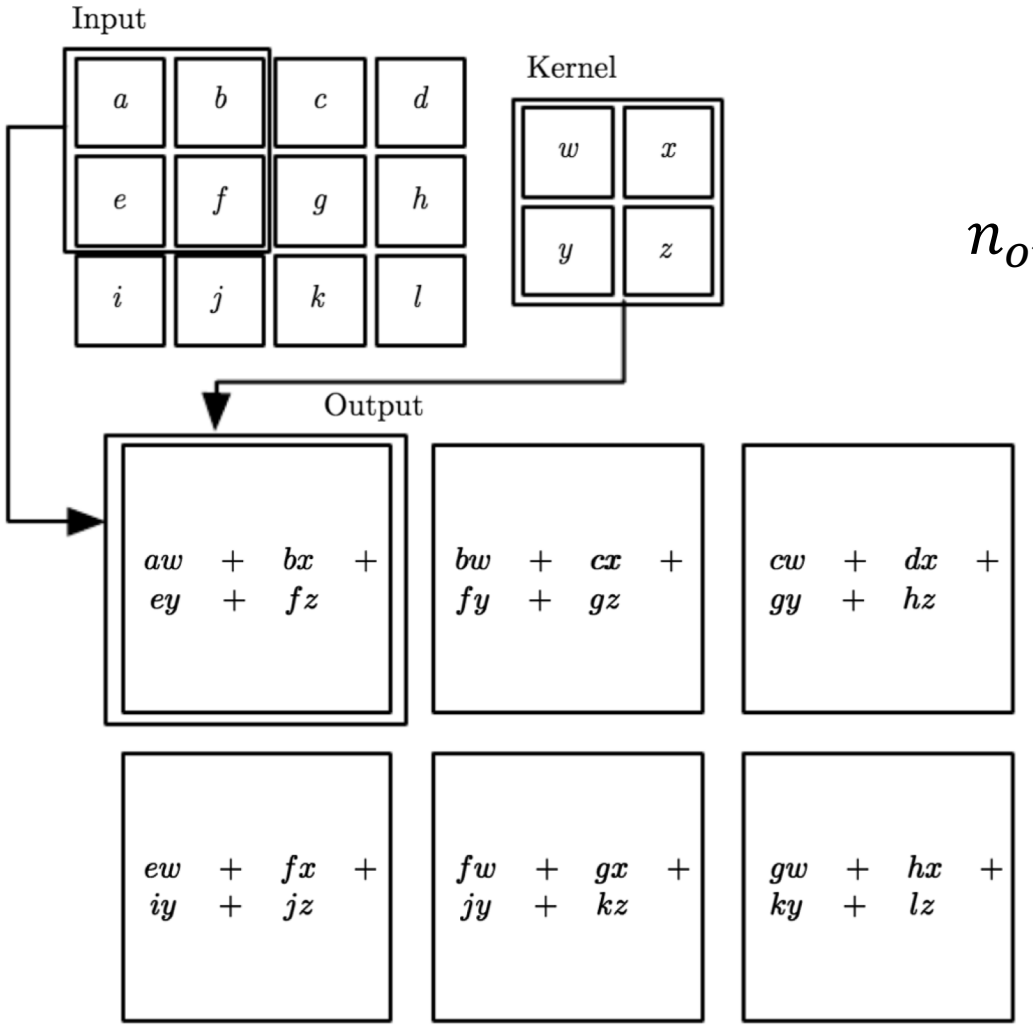
A neuron located in row i, column j of a given layer is connected to the outputs of the neurons in the previous layer located in rows i to $i + f_h - 1$, columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field. In order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called **zero padding**.



It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields. The distance between two consecutive receptive fields is called the **stride**. In the diagram, a 5 × 7 input layer (plus zero padding) is connected to a 3 × 4 layer, using 3 × 3 receptive fields and a stride of 2 (in this example the stride is the same in both directions, but it does not have to be so). A neuron located in row i, column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$, columns $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

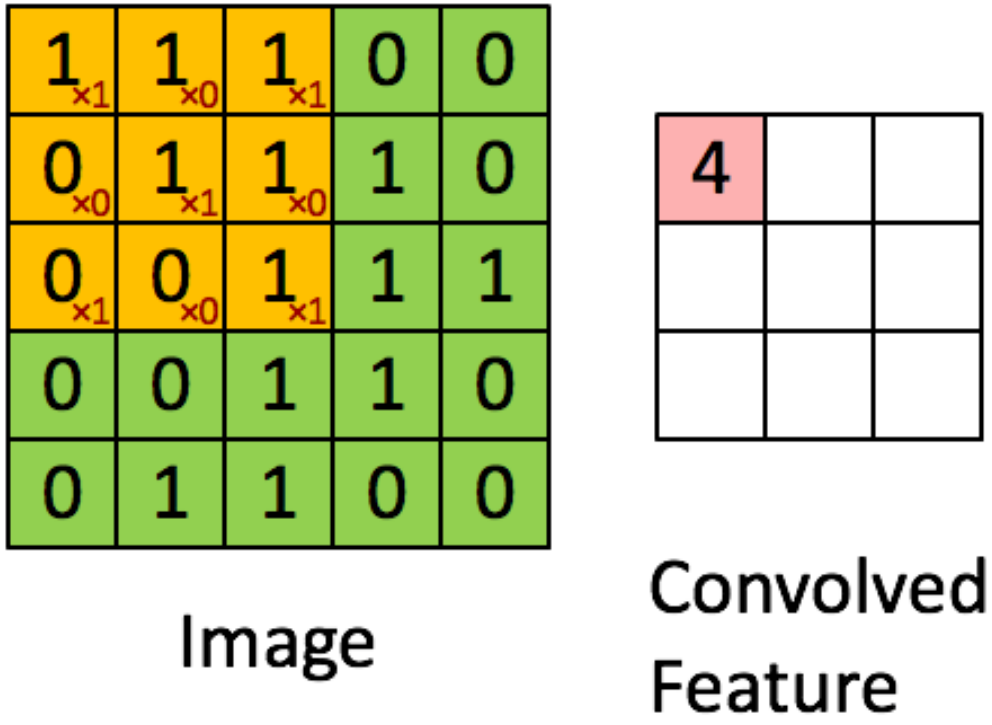


Convolutional Filters

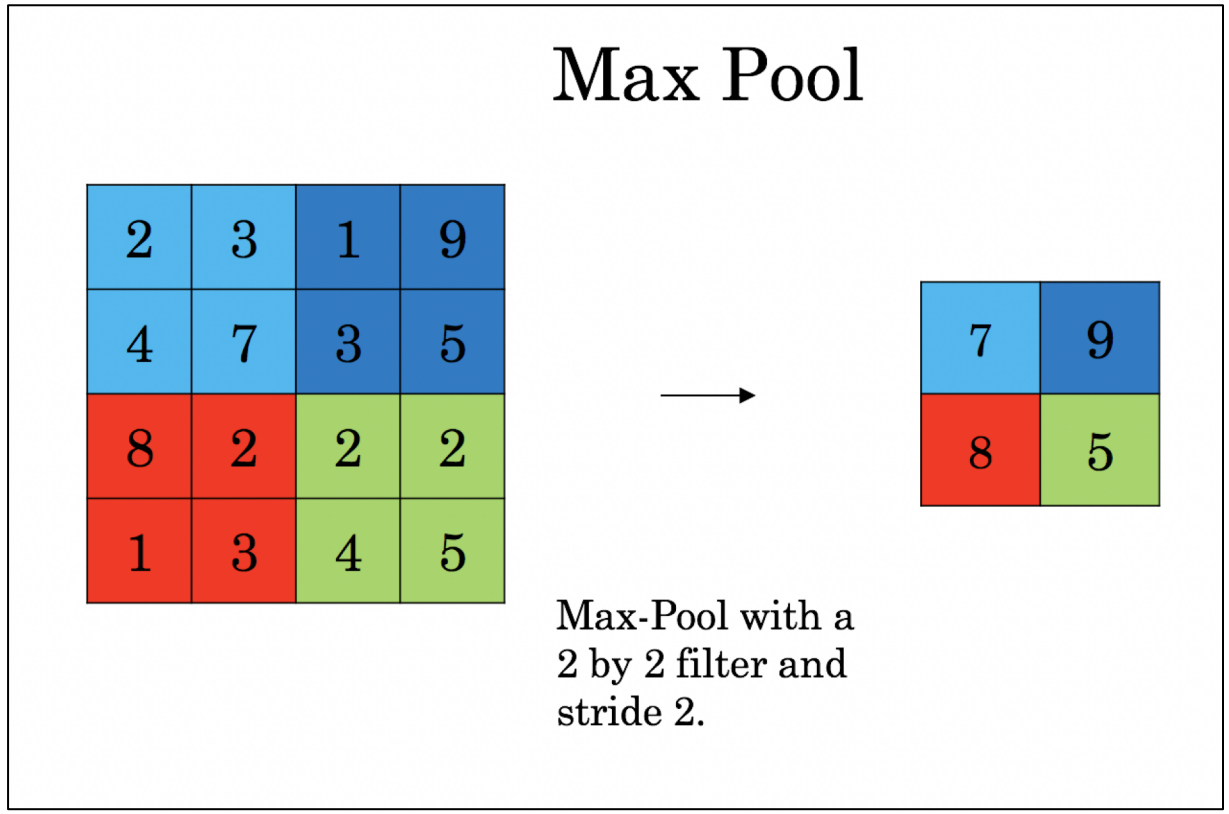


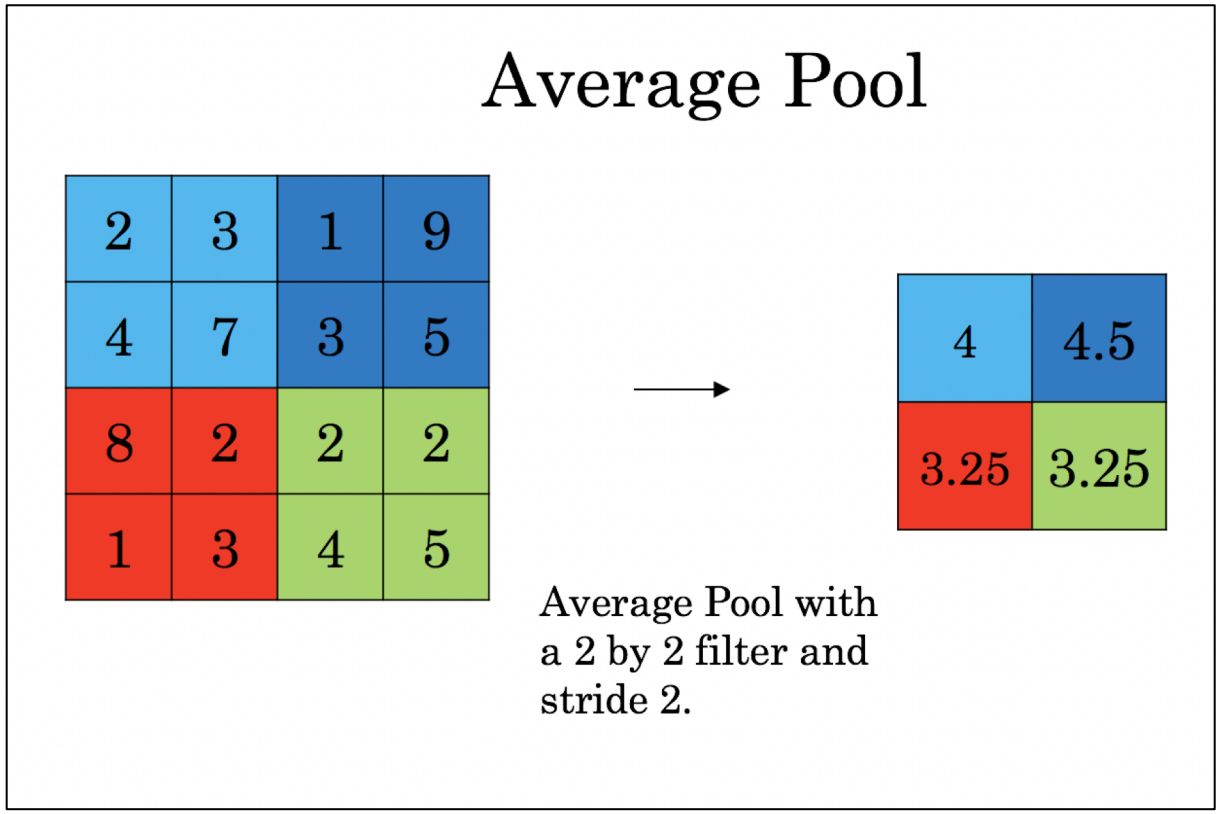
$$n_{out} = \left\lfloor \frac{n_{input} + 2 * padding - filter}{stride} \right\rfloor + 1$$

Convolutional filters



Pooling


Max Pool
2 3 1 9
4 7 3 5
8 2 2 2
1 3 4 5
7 9
8 5
Max-Pool with a 2 by 2 filter and stride 2.


Average Pool
2 3 1 9
4 7 3 5
8 2 2 2
1 3 4 5
4 4.5
3.25 3.25
Average Pool with a 2 by 2 filter and stride 2.

Pooling

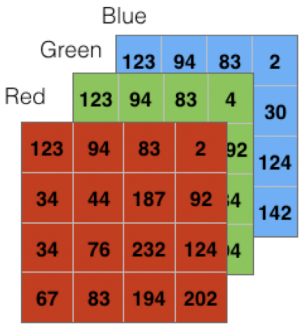
- The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input.
- Pooling does not require weights

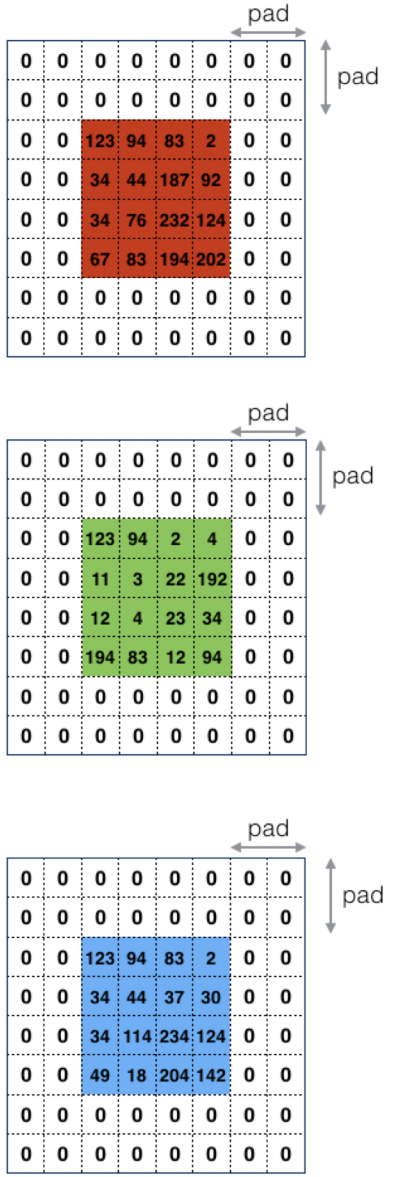
$$n_{out} = \left\lfloor \frac{n_{input} - filter}{stride} \right\rfloor + 1$$

Padding

- It allows you to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.

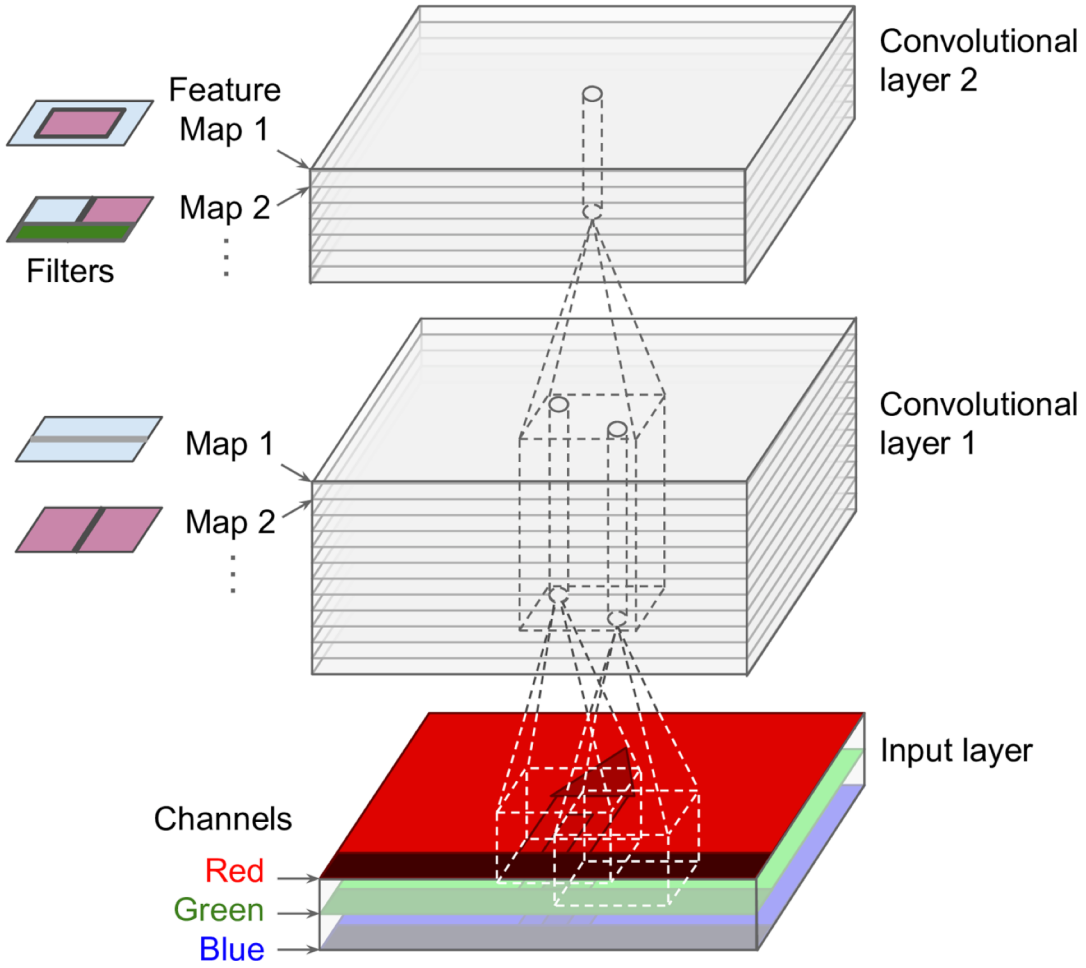
=





Convolution layers with multiple feature maps, and images with three channels

A neuron located in row *i*, column *j* of the feature map *k* in a given convolutional layer *l* is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). Note that all neurons located in the same row *i* and column *j* but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.



Computing the output of a neuron in a convolutional layer

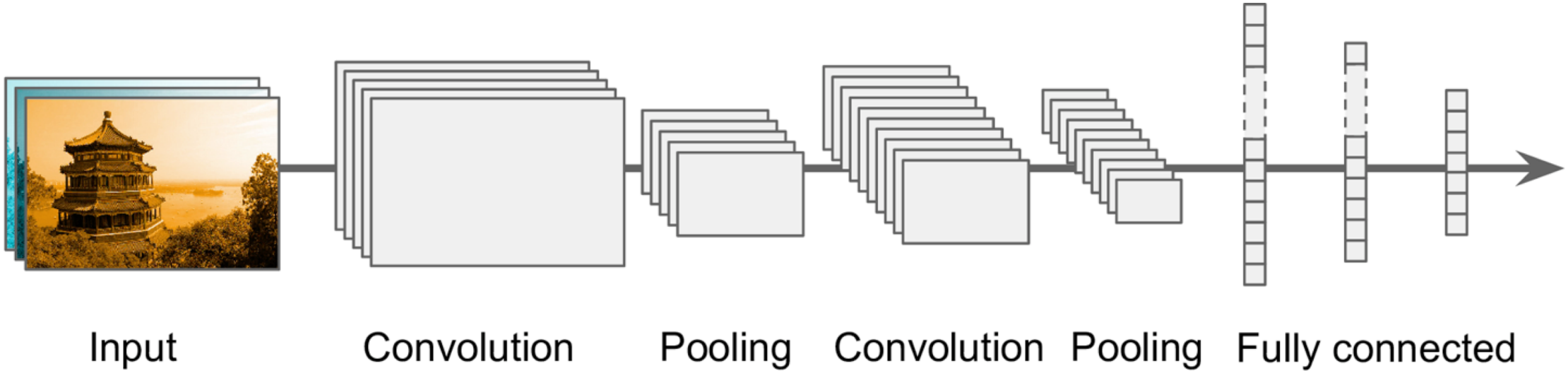
$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} . w_{u,v,k',k} \quad \text{with} \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i, j, k}$ is the output of the neuron located in row *i*, column *j* in feature map *k* of the convolutional layer (layer *l*).
- As explained earlier, s_h and s_w are the vertical and horizontal strides, f_h and f_w are the height and width of the receptive field, and $f_{n'}$ is the number of feature maps in the previous layer (layer *l* – 1).
- $x_{i', j', k'}$ is the output of the neuron located in layer *l* – 1, row *i*′, column *j*′, feature map *k*′ (or channel *k*′ if the previous layer is the input layer).
- b_k is the bias term for feature map *k* (in layer *l*). You can think of it as a knob that tweaks the overall brightness of the feature map *k*.
- $w_{u, v, k', k}$ is the connection weight between any neuron in feature map *k* of the layer *l* and its input located at row *u*, column *v* (relative to the neuron's receptive field), and feature map *k*′.

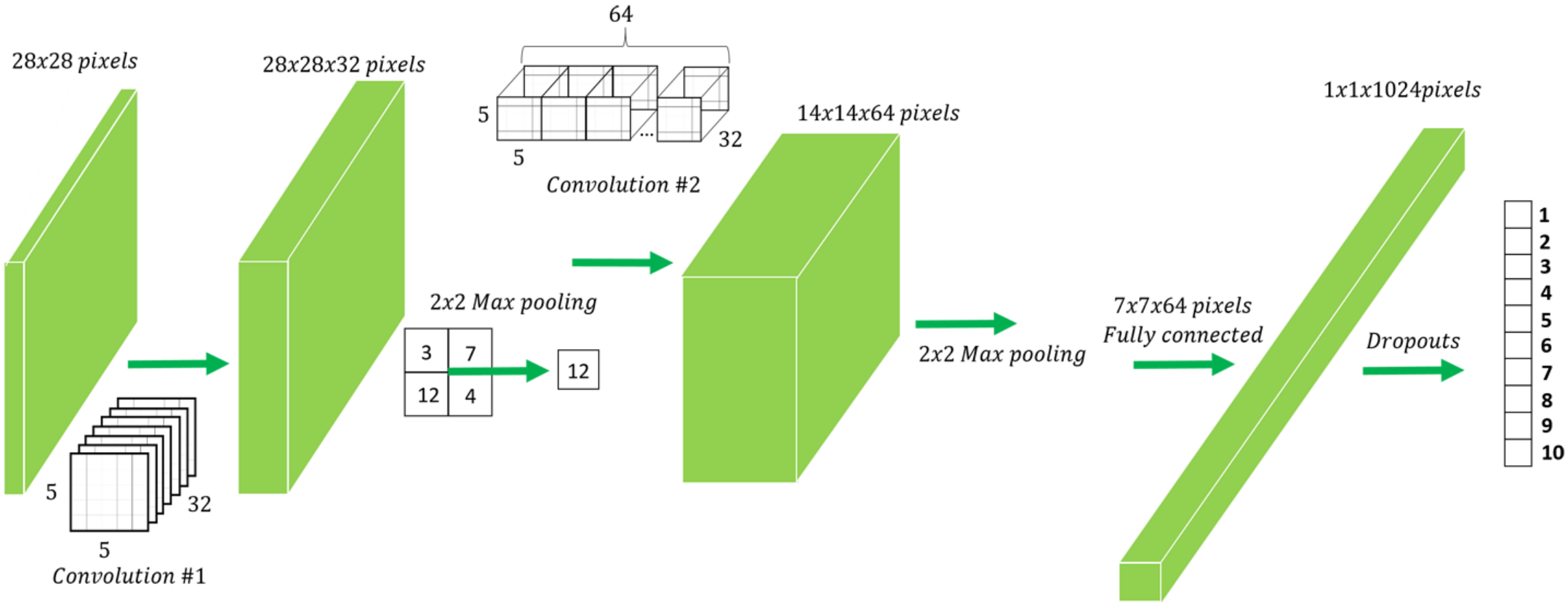
TensorFlow Implementation

Each input image is typically represented as a 3D tensor of shape [height, width, RGB channels]. A mini-batch is represented as a 4D tensor of shape [mini-batch size, height, width, channels]. The weights of a convolutional layer are represented as a 4D tensor of shape $[f_h, f_w, f_{n'}, f_n]$. The bias terms of a convolutional layer are simply represented as a 1D tensor of shape $[f_n]$.

CNN Architecture



LeNet-5 architecture for mnist dataset



CNN Architecture

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps) thanks to the convolutional layers At the top of the stack, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

A common mistake is to use convolution kernels that are too large. You can often get the same effect as one convolutional layer with a 9 × 9 kernel by stacking two layers with 3 × 3 kernels, for a lot less compute and parameters.

Classic CNN

- LeNet-5 (~ 60K parameters)
- Alex Net (~ 60M parameters)
- VGG-16 (~128M parameters)
- Resnet (depth - 152)

LeNet-5 for MINIST

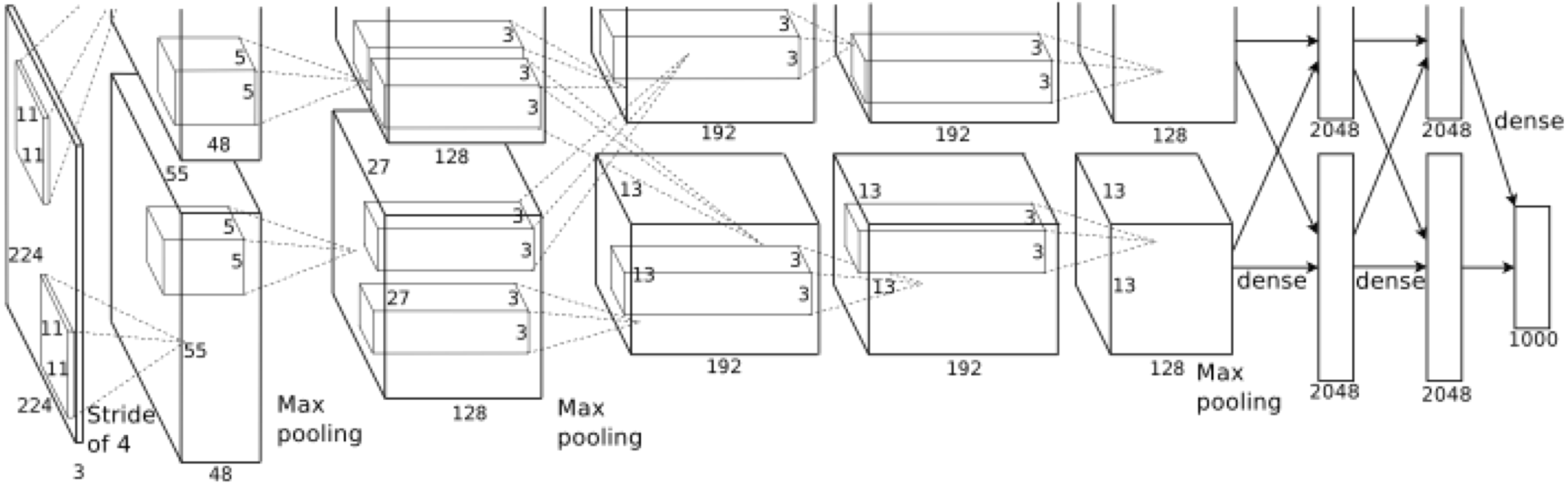
Total number of parameters = ~60K

Layer	Type	Maps/ Channels	Filter/Kernel	Stride	Size	Activation
In	Input	1	-	-	32x32x1	-
C1	Convolution	6	5x5	1	28x28x6	tanh
S2	Average Pooling	6	2x2	2	14x14x6	tanh
C3	Convolution	16	5x5	1	10x10x16	tanh
S4	Average Pooling	16	2x2	2	5x5x16	tanh
c5	Convolution	120	2x2	1	1x120	tanh
F6	Fully Connected	-	-	-	84	tanh
Out	Fully Connected	-	-	-	10	RBF

AlexNet

```
network = input_data(shape=[None, 227, 227, 3])
network = conv_2d(network, 96, 11, strides=4, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
network = conv_2d(network, 256, 5, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
network = conv_2d(network, 384, 3, activation='relu')
network = conv_2d(network, 384, 3, activation='relu')
network = conv_2d(network, 256, 3, activation='relu')
network = max_pool_2d(network, 3, strides=2)
network = local_response_normalization(network)
network = fully_connected(network, 4096, activation='tanh')
network = dropout(network, 0.5)
network = fully_connected(network, 4096, activation='tanh')
network = dropout(network, 0.5)
network = fully_connected(network, 17, activation='softmax')
```

Alexnet


224
224
3
11
11
Stride
of 4
55
48
5
5
27
128
3
3
3
Max
pooling
13
192
192
128
Max
pooling
2048
2048
dense
dense
dense
1000
2048
2048

Why not use fully connected NN?

- Unfortunately, although fully connected neural network works fine for small images (e.g., MNIST), it breaks down for larger images because of the huge number of parameters it requires.
- For example, a 100 × 100 image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means a total of 10 million connections. And that's just the first layer. CNNs solve this problem using partially connected layers.

Image classification datasets

Name	Image Size	Dataset Size	Classes	Description
MNIST	28x28x1	60,000, 10,000	10	Handwritten digits
CIFAR10	32x32x3	50,000, 10,000	10	Single objects like cat, bird, airplane etc.
CIFAR100	32x32x3	50,000, 10,000	100	Single objects like beaver, dolphin, otter, seal etc.
STL10				
SVHN		600,000 +		Numbers from Google Street View
Oxford Flower		1360	17	17 common flowers in UK
EMNIST		814,255		Handwritten digits or characters from 500 writers
ImageNet		13 million	10000	

https://medium.com/startup-grind/fueling-the-ai-gold-rush-7ae438505bc2

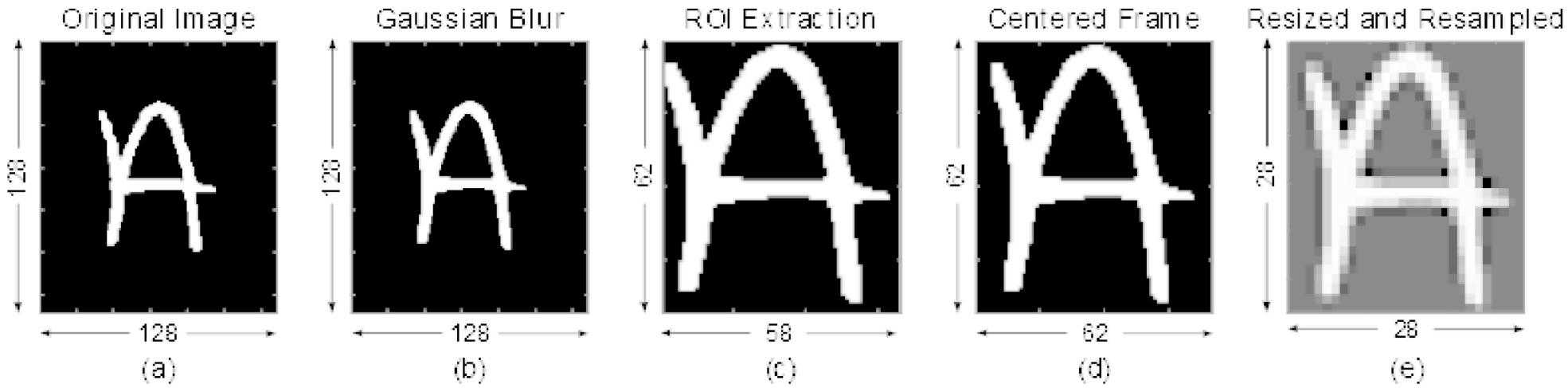
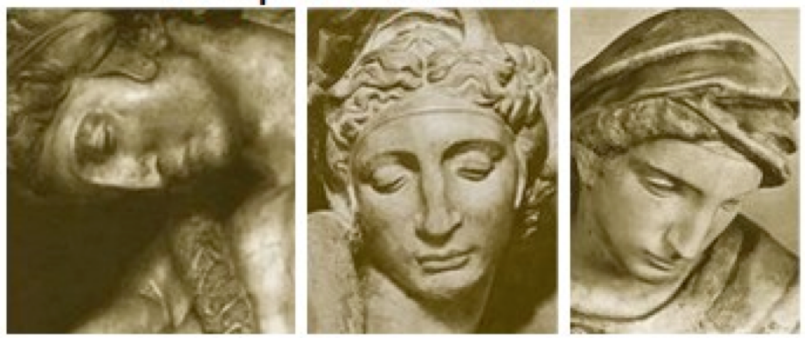


Fig. 1. **Diagram of the conversion process used to convert the NIST dataset.** The original images are stored as 128×128 pixel binary images as shown in (a). A Gaussian filter with $\sigma = 1$ is applied to the image to soften the edges as shown in (b). As the characters do not fill the entire image, the region around the actual digit is extracted (c). The digit is then placed and centered into a square image (d) with the aspect ratio preserved. The region of interest is padded with a 2 pixel border when placed into the square image, matching the clear border around all the digits in the MNIST dataset. Finally, the image is down-sampled to 28×28 pixels using bi-cubic interpolation. The range of intensity values are then scaled to $[0, 255]$, resulting in the 28×28 pixel gray-scale images shown in (e).

Example: image processing

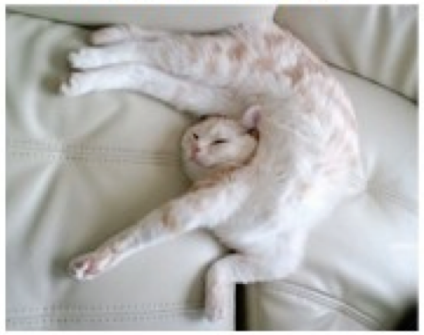
Challenges in image processing

Viewpoint variation



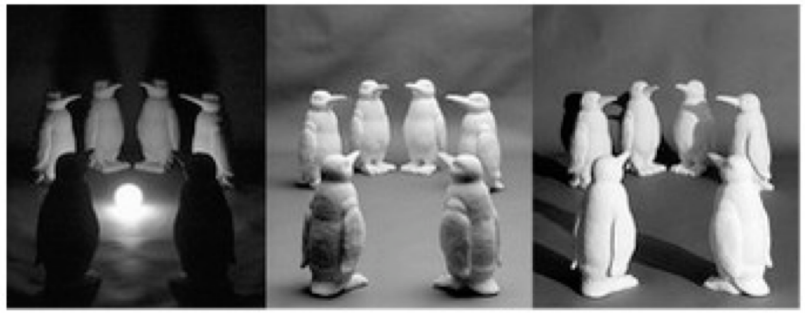
Scale variation

Deformation



Occlusion

Illumination conditions



Background clutter

Intra-class variation

Sequence Learning

Reference

A Critical Review of Recurrent Neural Networks for Sequence Learning

Zachary C. Lipton
zlipton@cs.ucsd.edu

John Berkowitz
jaberkow@physics.ucsd.edu

Charles Elkan
elkan@cs.ucsd.edu

June 5th, 2015

https://arxiv.org/pdf/1506.00019.pdf

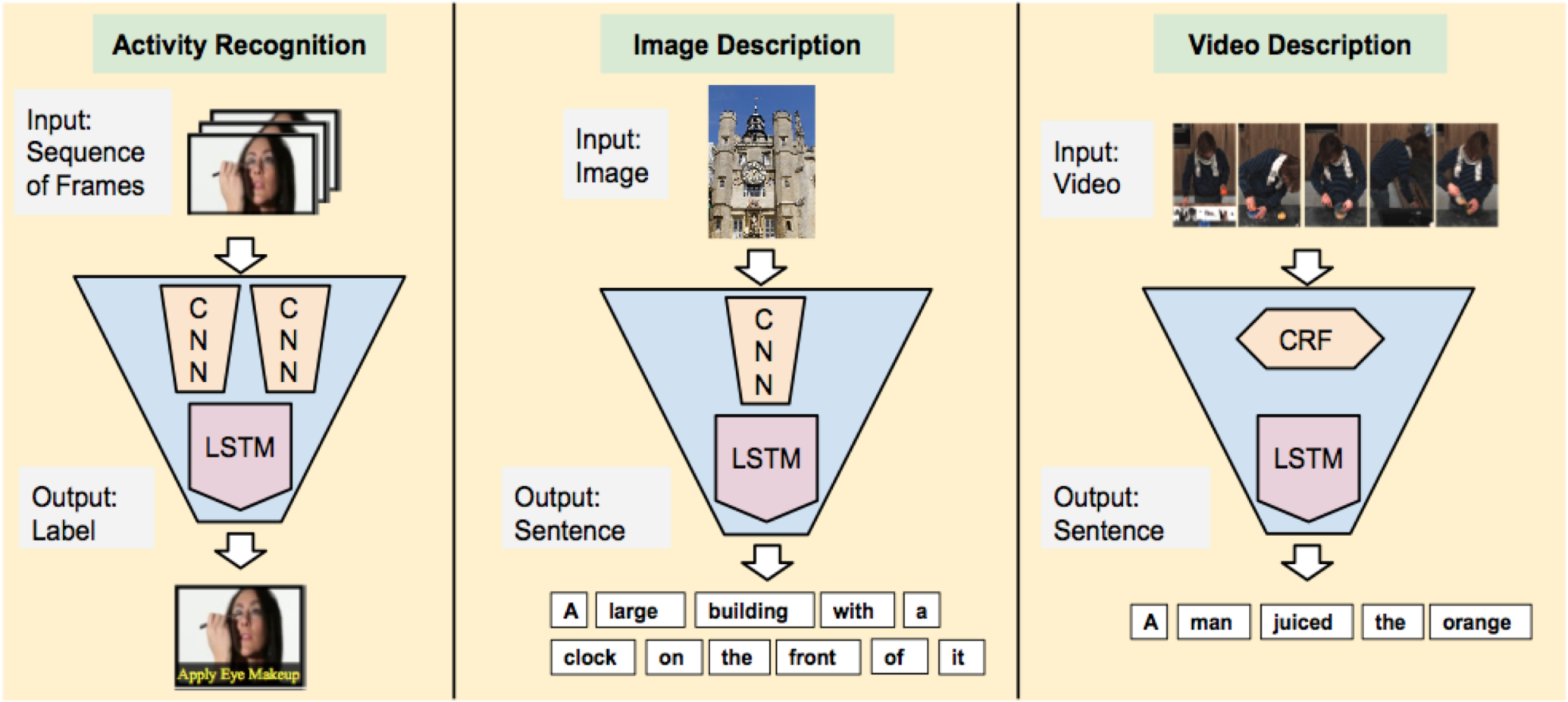
Examples of sequence data

Speech recognition		→	"The quick brown fox jumped over t lazy dog."
Music generation	∅	→	
Sentiment classification	"There is nothing to like in this movie."	→	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG	→	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi?	→	Do you want to sing with me?
Video activity recognition		→	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger.	→	Yesterday, Harry Potter met Hermione Granger.

Types of architecture

- Many to many ($T_x > 1$ and $T_y > 1$)
  - Example: language translation ($T_x \neq T_y$), NER ($T_x = T_y$)
- Many to one ($T_x > 1$, $T_y = 1$)
  - Sentiment classifier
- One to many ($T_x = 1$, $T_y > 1$)
  - Music generation

Visual Sequence Tasks



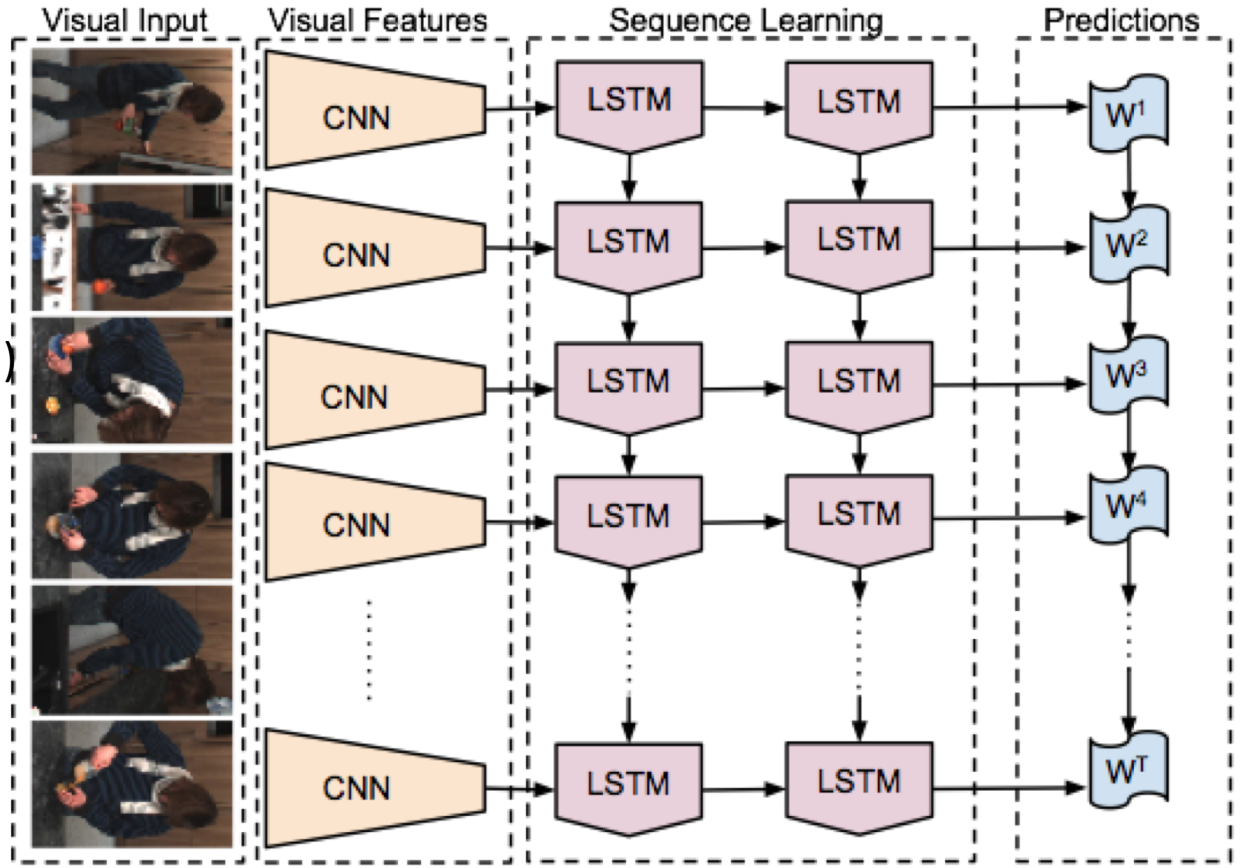
Recurrent Networks for Sequences

LRCN: Long-term Recurrent Convolutional Network

- activity recognition (sequence-in)
- image captioning (sequence-out)
- video captioning (sequence-to-sequence)

Recurrent Nets and Long Short Term Memories (LSTM) are sequential models

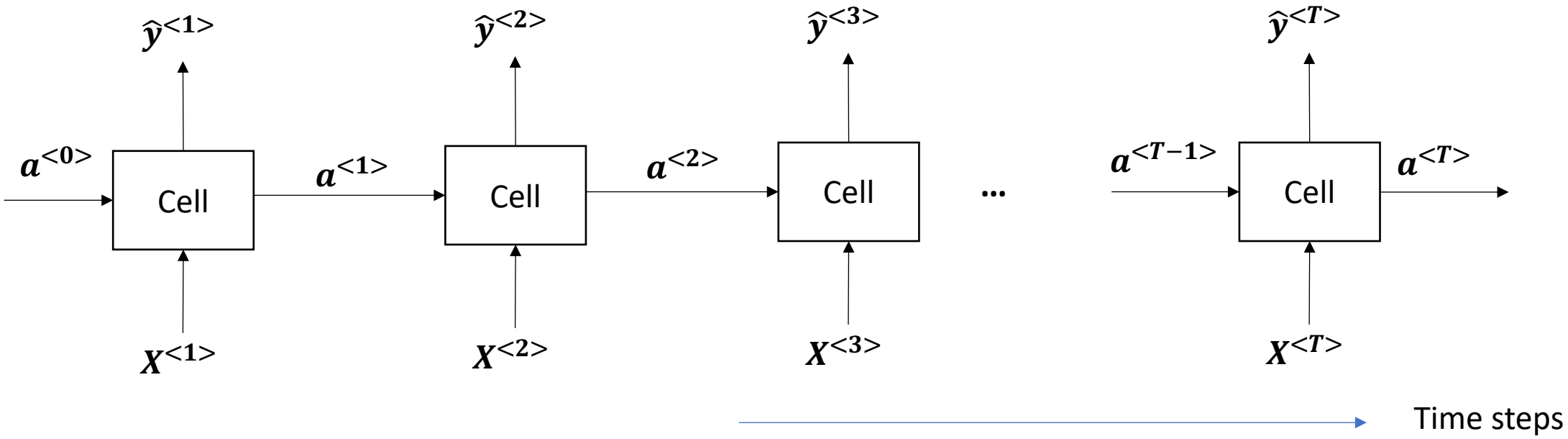
- video
- language
- dynamics



Use-cases of RNN

- RNN have been extensively used by NLP community for various applications
  - Build language model - predict the probability of the next word given the sequence of previous words
  - Important for various higher level tasks like machine translation and spelling correction, speech to text, sentiment analysis, text generation
- Analyze time series data such as stock prices, and tell you when to buy or sell
- In autonomous driving systems, anticipate car trajectories and help avoid accidents
- Image captioning

Basic architecture of RNN

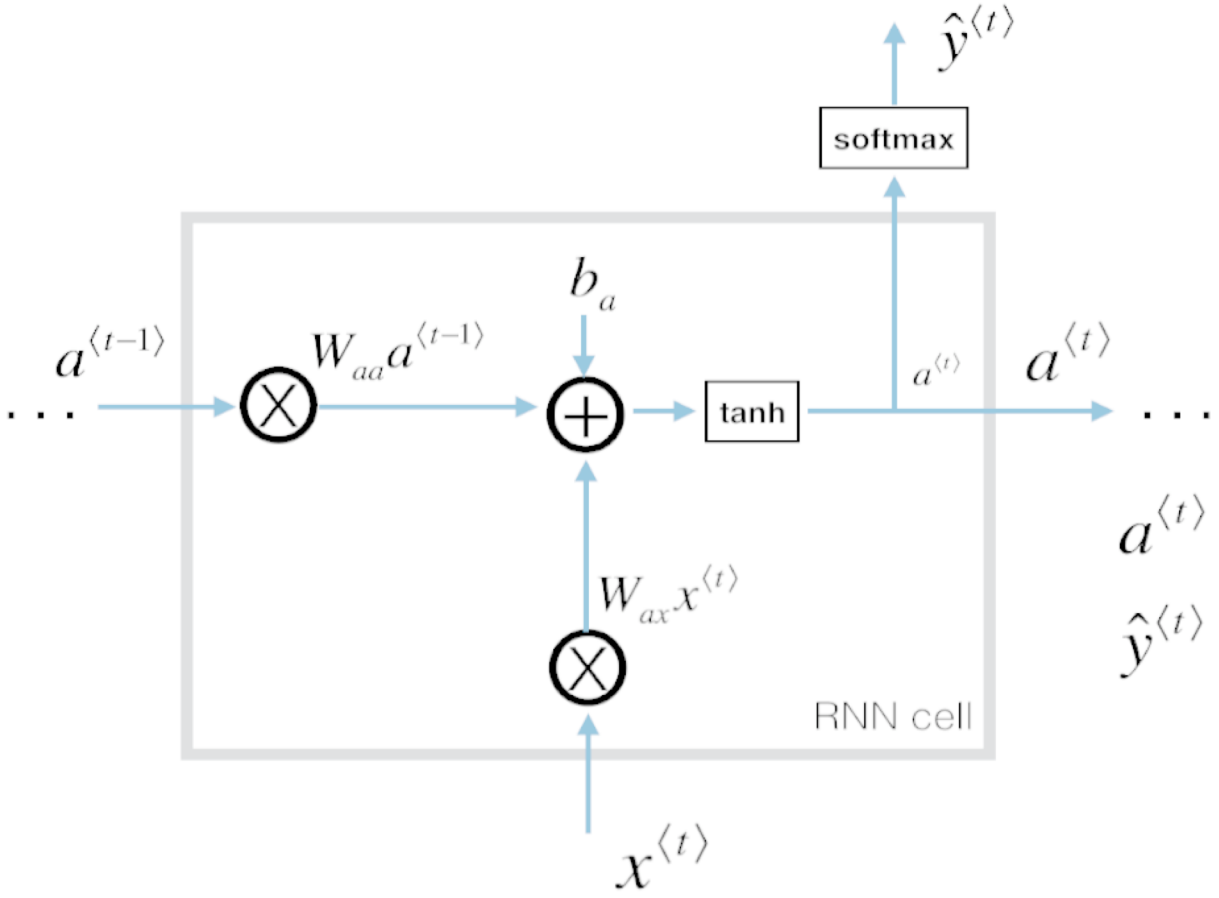


- $a^{<0>}$ is the initial state. Often it is set to 0
- $a^{<t>}$ is called state. Dimension of $a^{<t>}$ is called state size or hidden size.
- T is called max sequence length. If your actual input is not as long as max sequence length or longer (consider a text where number of words is not fixed), you have to pad or truncate.

Simple RNN Cell

- All inputs are independent of each other
- RNN cells incorporate dependence by having state or memory
- Value of the hidden state at a moment is a function of
  - Value of hidden state at previous time step
  - Value of the input at current time step
- Unrolling - draw network out for complete sequence

RNN cell - also called memory cell



$$W_{ax}$$

Indicates this W matrix will be multiplied with X

Indicates this W matrix will be used to produce a like quantity

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

RNN cell - simplified expressions

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}X^{<t>} + b_a)$$

$$= g(W_a[\begin{matrix} a^{<t-1>} \\ X^{<t>} \end{matrix}] + b_a) \quad \text{where } W_a = [W_{aa}\, W_{ax}]$$

Row wise stacking

Column wise stacking

$$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$$

Total parameters := $(n_x + n_a + 1) * n_a + (n_a + 1) * n_y$

Example:

$a^{<t-1>}$: $(100,)$
$X^{<t>}$: $(10000,)$
W_{ax} : $(100, 10000)$
W_{aa} : $(100, 100)$
W_a : $(100, 10100)$

Shapes

W_a : $(n_a, n_a + n_x)$
b_a : n_a
W_y : (n_a, n_y)
b_y : n_y

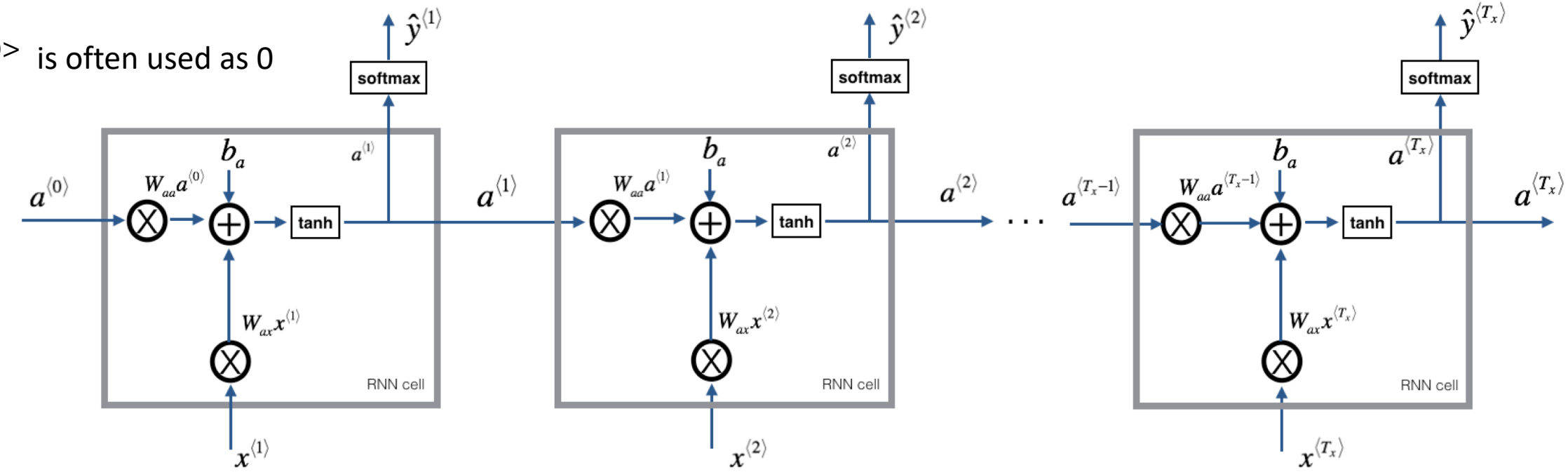
n_a: size of the state

Notations

- Superscript [l] denotes an object associated with the l^{th} layer.
  - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object associated with the i^{th} example.
  - Example: $x^{(i)}$ is the i^{th} training example input.
- Superscript $\langle t \rangle$ denotes an object at the t^{th} time-step.
  - Example: $x^{\langle t \rangle}$ is the input x at the t^{th} time-step. $x^{(i)\langle t \rangle}$ is the input at the t^{th} timestep of example i.
- Lowerscript i denotes the i^{th} entry of a vector.
  - Example: $a_i^{[l]}$ denotes the i^{th} entry of the activations in layer l

RNN (forward pass)

$a^{<0>}$ is often used as 0



$$a^{<t>} = g(w_{aa}a^{<t-1>} + w_{ax}X^{<t>} + b_a)$$

g: tanh is commonly used

$$\hat{y}^{<t>} = g(w_{ya}a^{<t>} + b_y)$$

g: sigmoid or softmax is commonly used

RNN Loss

$$\mathcal{L}^{<t>} = \ softmax\ cross\ entropy\ for\ tth\ time\ sequence$$

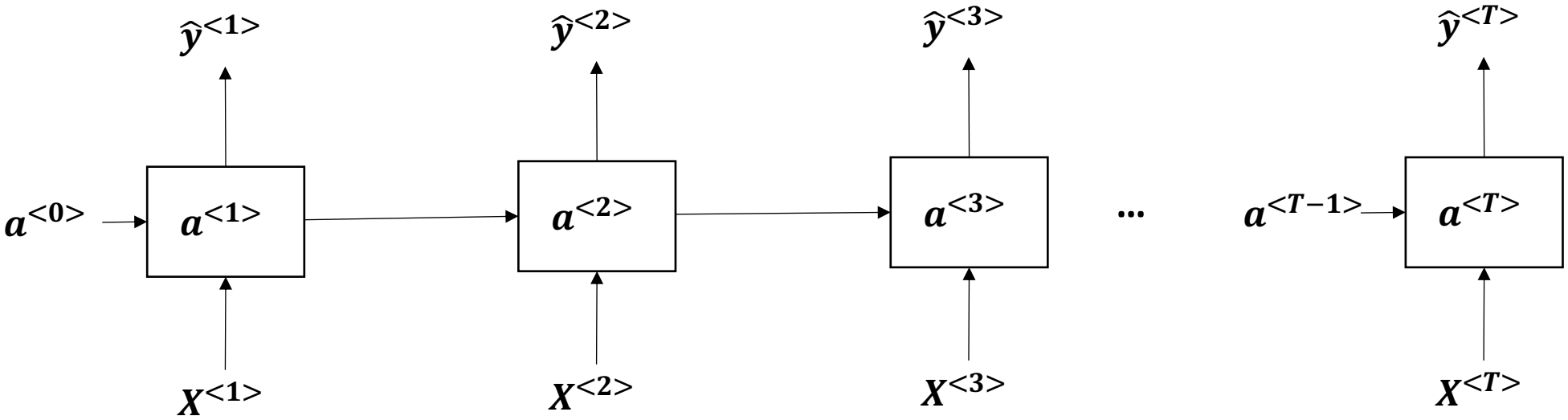
$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}^{<t>}$$

Language modeling: word vs character

- In word based RNN, we calculate the probability of the next work given the sequence of previous words.
- In character based RNN, we estimate the probability of next based on the previous characters. Character based models needs less amount data to train the model compared to word based models.

Vanishing and exploding gradients

For very deep (for example, 100 layers) the magnitude of back propagation becomes weak because of the exponential nature of the gradient. It is also possible that gradient increasingly becomes larger leading to exploding gradient problem. One possible solution of exploding gradient is gradient clipping. More generic solution is GRU (Gated Recurrent Unit) or LSTM (Long Short Term Memory).



Example of long range dependencies:

The cat, which already ate pies **was** full
The cats, which already ate pies **were** full

Simplified GRU (Gated Recurrent Unit)

Column-wise stacking

$$\tilde{c}^{<t>} = tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$$

$$\mathbf{\Gamma}_u{}^{<t>} = sigmoid(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$c^{<t>} = \mathbf{\Gamma}_u * \tilde{c}^{<t>} + (1 - \mathbf{\Gamma}_u) * c^{<t-1>}$$

Element wise multiplication

$c^{<t>}$ is similar to $a^{<t>}$ as discussed in RNN

Update gate balancing the between current timestamp cell and previous timestamp cell value

Other form of GRU

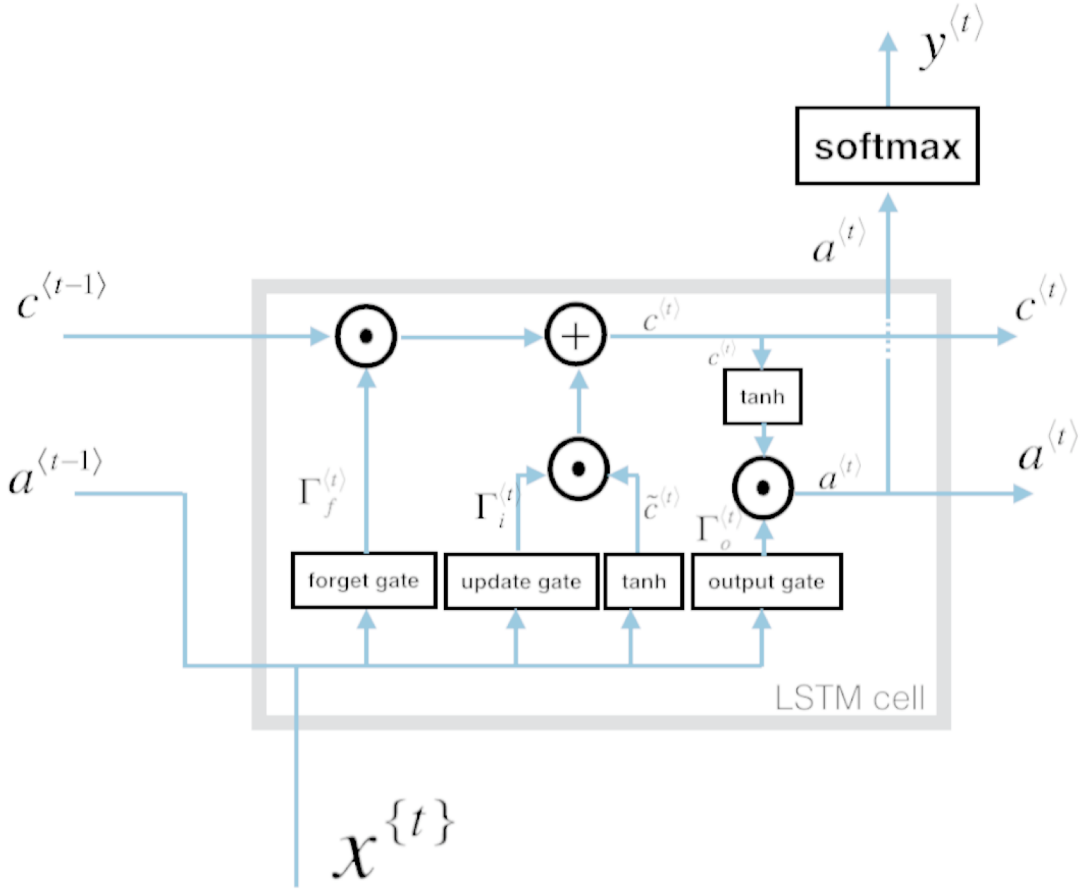
$$\mathbf{\Gamma}_r^{<t>} = sigmoid(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$ Reset gate

$$\tilde{c}^{<t>} = tanh(W_c[\mathbf{\Gamma}_r^{<t>} * c^{<t-1>}, x^{<t>}] + b_c)$$ Traditional recurrent unit

$$\mathbf{\Gamma}_u^{<t>} = sigmoid(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$ Update gate control

$$c^{<t>} = \mathbf{\Gamma}_u * \tilde{c}^{<t>} + (1 - \mathbf{\Gamma}_u) * c^{<t-1>}$$

LSTM (Long short term memory) Cell



. . .

$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$

$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$

$$\tilde{c}^{\langle t \rangle} = \tanh(W_c[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_c)$$

$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$

$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$

$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

$$y^{<t>} = softmax(W_y a^{<t>} + b_y)$$

Total parameters = **4** * (n_x + n_a + 1) * n_a + (n_a + 1) * n_y

LSTM

The LSTM has the ability to remove or add information to the cell state, carefully regulated by structures called gates.

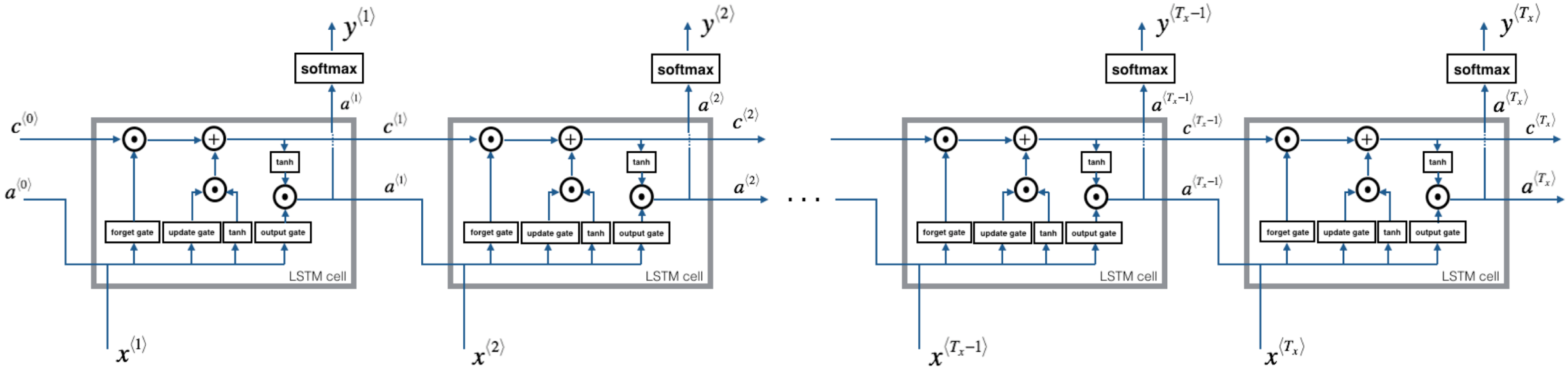
Forget gate: Lets assume we are reading words in a piece of text, and want use an LSTM to keep track of grammatical structures, such as whether the subject is singular or plural. If the subject changes from a singular word to a plural word, we need to find a way to get rid of our previously stored memory value of the singular/plural state.

Update Gate: Once we forget that the subject being discussed is singular, we need to find a way to update it to reflect that the new subject is now plural

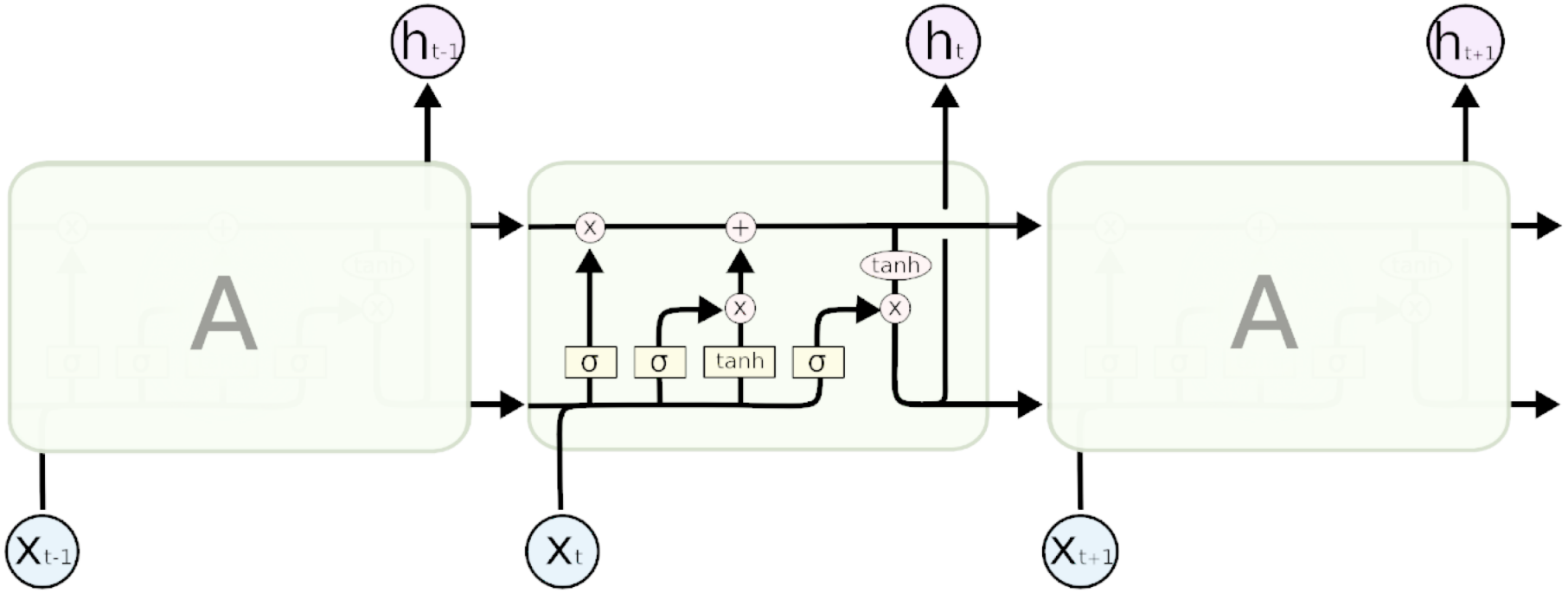
Updating the cell: To update the new subject we need to create a new vector of numbers that we can add to our previous cell state

Output Gate: To decide which outputs we will use

LSTM - forward pass

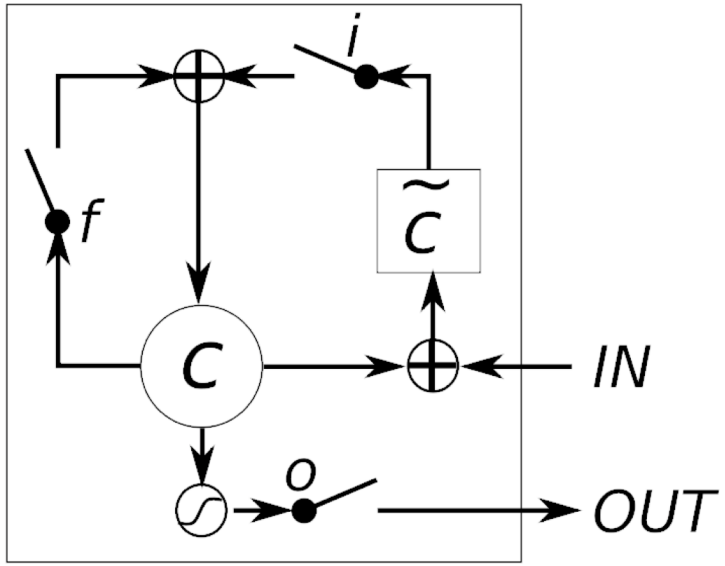


LSTM

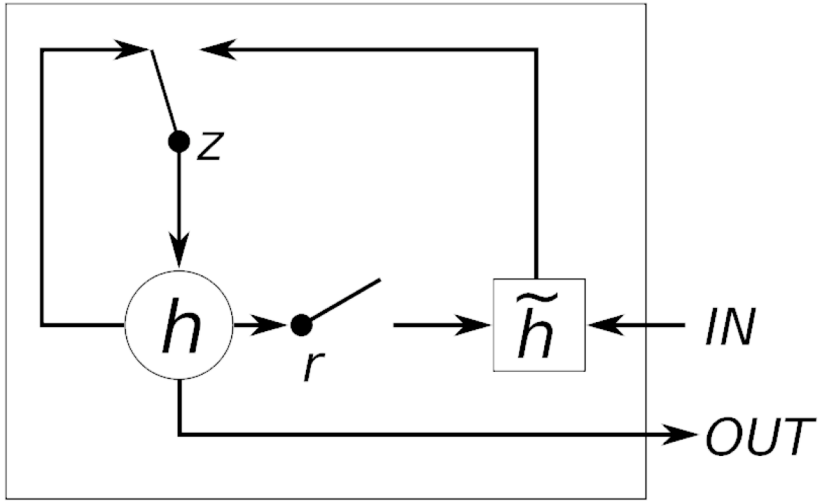


http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM vs GRU



(a) Long Short-Term Memory



(b) Gated Recurrent Unit

https://arxiv.org/pdf/1412.3555.pdf

Limitations of RNN

To produce $y^{<t>}$ only $X^{<1>}, X^{<2>}$ … $X^{<t-1>}$, $X^{<t>}$ inputs are used.
However if we look further in the time sequence, that might give vital clues.

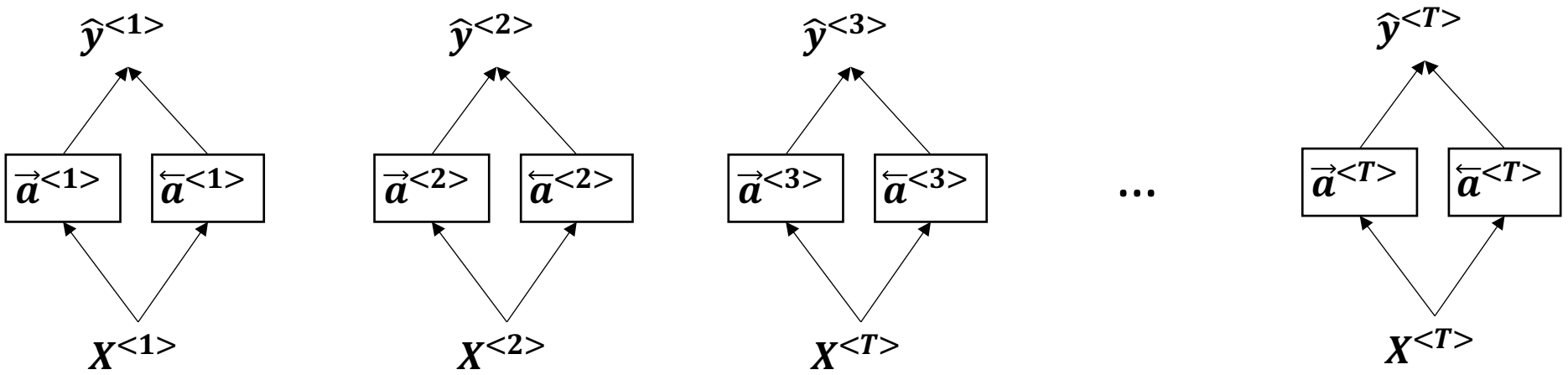
For example, consider the following NER problem - we want to know whether a word in the sentence indicate name of the person.

He said “**Teddy** Roosevelt was a great president”
He said “**Teddy** bears are on sale”

Solution to this problem is to use bi-directional neural networks (BRNN)

Bidirectional RNN

$$\hat{y}^{<T>} = softmax(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$



Memory cell can be of GRU/LSTM etc. type

Text Processing

Dataset	Source	Labels	Statistical Models	CNN
Flipkart Twitter Sentiment	Twitter	Pos, Neg	85%	96%
Flipkart Twitter Sentiment	Twitter	Pos, Neg, Neu	76%	89%
Fine grained sentiment in Emails	Emails	Angry, Sad, Complaint, Request	55%	68%
SST2	Movie Reviews	Pos, Neg	79.4%	87.5%
SemEval Task 4	Restaurant Reviews	food / service / ambience / price / misc	88.5%	89.6%

https://youtu.be/04ev55WnvSg?t=27m31s

Wordnet in python

```
from nltk.corpus import wordnet as wn
wn.synsets('dog')
```

```
[Synset('dog.n.01'),
 Synset('frump.n.01'),
 Synset('dog.n.03'),
 Synset('cad.n.01'),
 Synset('frank.n.02'),
 Synset('pawl.n.01'),
 Synset('andiron.n.01'),
 Synset('chase.v.01')]
```

http://www.nltk.org/howto/wordnet.html

Word2Vec

- Created by a team of researchers led by Tomas Mikolov at Google
- If the user searches for [Dell notebook battery size], we would like to match documents with [Dell laptop battery capacity]
- One way to solve the above problem would be to dictionary of synonyms. nltk wordnet is popular go to approach - not practically useful in most circumstances.
- Another approach is a word2vec - which encode the word by a vector that represents a word by means of its neighbors.
- “You shall know a word by the company it keeps”

Discrete Representation

- In vector space terms each word is presented by by a dense vector of 1 and 0 like [0 0 0 … 1 0 0]
- Dimensionality: 20k (speech), - 50 K (PTB), 500K (big vocab), 13M (Google 1T)
- This is known as one-hot representation
- It is a localist representation
- One problem - one hot encoded data do not reveal simiarity measures - each pair are orthogonal

Word2vec

- Build a dense vector for each word, chosen so that it is good at predicting other words appearing in its context
- It helps identify distributional similarity between a pair of words
- A model aims to predict between a center word w_t and the context words in terms of word vectors. p(context | w_t) = …
- Loss function, J = 1 - p(w_{-t} | w_t), w_{-t}: every word except t
- Predict between every word and its context words

Word2vec algorithms

- Skip-grams (SG) - predict context words given target (position independent)
- Contiguous bag of words (CBOW) - predict target word from bag of words context

- Two training methods
  - Hierarchical softmax
  - Negative sampling

Details of word2vec

- For each word t = 1 … T, predict surrounding words in a word of "radius" m of every word
- Objective function: maximize the probability of any context word given the current center word

$$J'(\theta) = \prod_{t=1}^{T} \prod_{-m \le j \le m, j \neq o} p\left(w_{t+j} \mid w_t; \theta\right)$$

Negative log likelihood

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \le j \le m, j \neq o} \log p\left(w_{t+j} \mid w_t\right)$$

θ represents all the variables we like to optimize

Details of word2vec

- Predict surrounding words in a window of radius m of every word

$$p(w_{t+j} \mid w_t) = p(o|c) = \frac{exp(u_o^T v_c)}{\sum_{w=1}^{V} exp(u_w^T v_c)}$$

Softmax form to convert vectors into probability

Where,
o is the outside (or output) word index, c is the center word index
v_c and u_o are center and outside vectors of indices c and o

Softmax using word c to obtain probability of word o

Generative models

A generative model describes how data is generated, in terms of a probabilistic model.

In the scenario of supervised learning, a generative model estimates the joint probability distribution of data $P(X, Y)$ between the observed data X and corresponding labels Y.

Examples of popular generative models are:

- Naive Bayes
- Hidden Markov Models
- Latent Dirichlet Allocation
- Boltzmann Machines

Generative model for classification

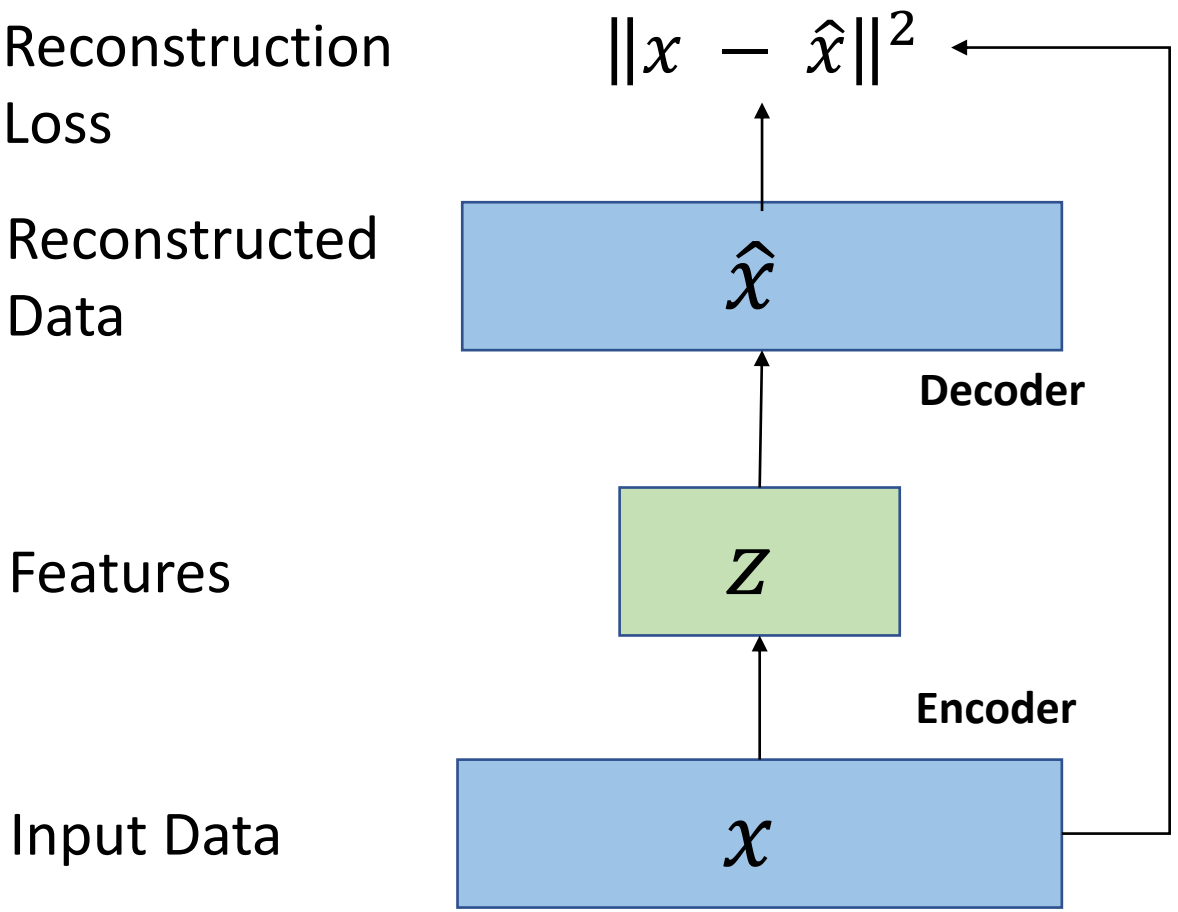
Class prediction = $argmax\ P(y\,|x) = argmax\ \frac{P(y,x)}{P(x)}$

Since the denominator is same for all classes,

Class prediction = $argmax\ \frac{P(y,x)}{P(x)}$

Auto Encoders

Auto encoders



Auto encoders are trained to represent data in lower dimension, called features such that the original can be reconstructed from the lower dimensional representation as accurately as possible.

What is auto encoders

Autoencoding is an unsupervised (self-supervised), data compression algorithm implemented in neural networks where the compression and decompression functions are

- data-specific
- lossy
- learned automatically from examples rather than engineered by a human.

Autoencoders are data-specific

It will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds.

Autoencoders are lossy

The decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

Autoencoders are learned by data

Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

What is required to train auto encoders?

To build an autoencoder, you need three things:

- an encoding function
- a decoding function
- a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function).

Use cases of autoencoders

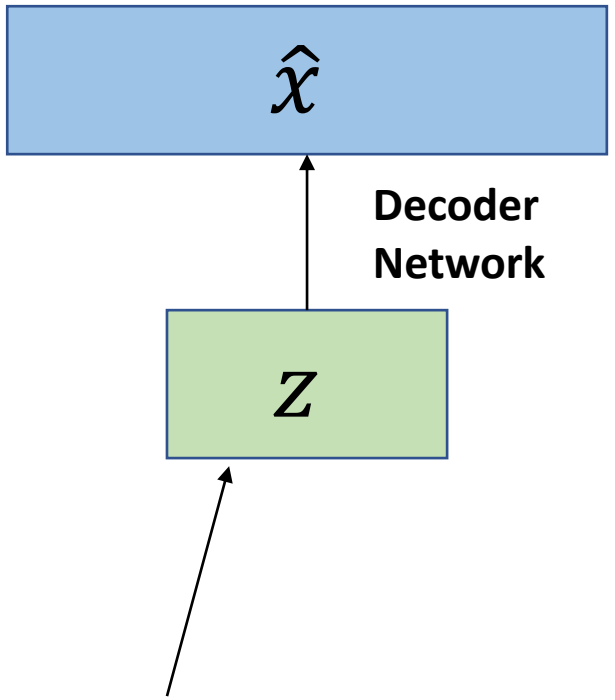
There are two popular use cases of autoencoders

- data denoising
- dimensionality reduction for data visualization. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

Variational auto encoder

Reconstructed Data

$p_\theta(x\,|z^{(i)})$



Samples from true prior

$p_\theta(z)$

This is also known as latent variable

We want to estimate θ* of this generative model.

How to training?

Maximize likelihood of training data

$$p_\theta(x) = \int p_\theta(z)p_\theta(x|z)dz$$

Simple Gaussian prior

Variational autoencoders

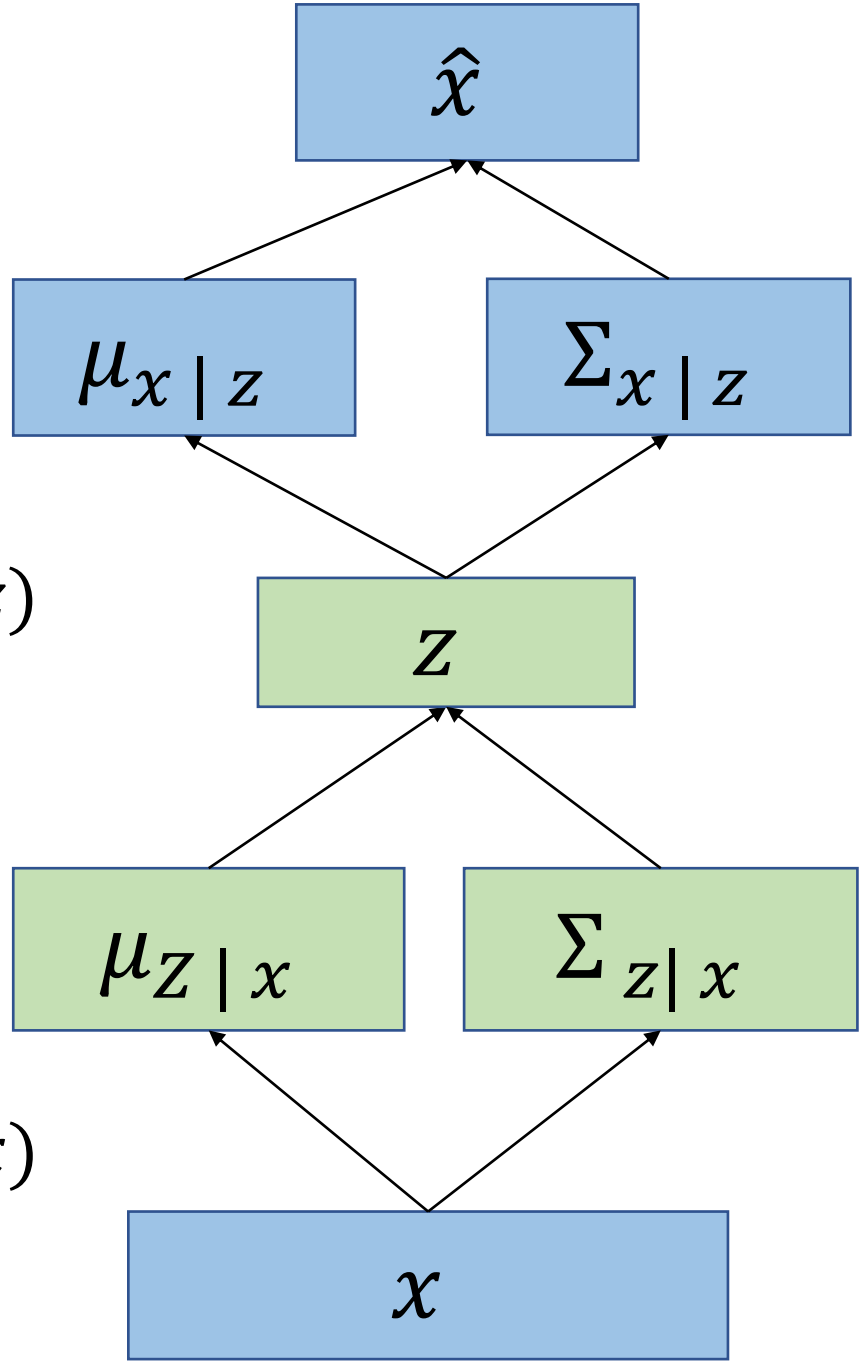
$$L(x^{(i)}, \theta, \varphi)$$
$$= E_z[\log p_\theta(x^{(i)} \mid z)] - D_{KL}(\theta_\varphi(z|x^{(i)}) \,||\, p_\varphi(z))$$

Decoder network, $p_\theta\ (x \mid z)$

$$z \mid x \sim \mathcal{N}(\mu_{z \mid x}, \Sigma_{z \mid x})$$

$$x \mid z \sim \mathcal{N}(\mu_{x \mid z}, \Sigma_{x \mid z})$$

Encoder network, $q_\varphi\ (z \mid x)$



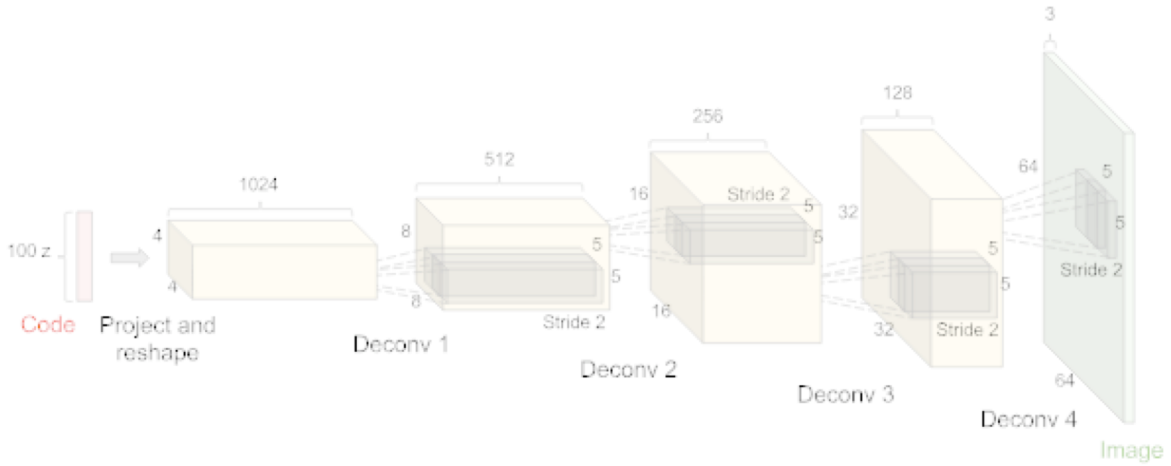
Training an auto encoder

Training VAE requires to solve two optimization problems

A. Minimize reconstruction loss - measured by cross entropy
B. Minimize latent loss = KL divergence between the encodings (z) and the target distribution - Gaussian normal. [KL divergence measures the difference between two distributions]

DCGAN - an example of

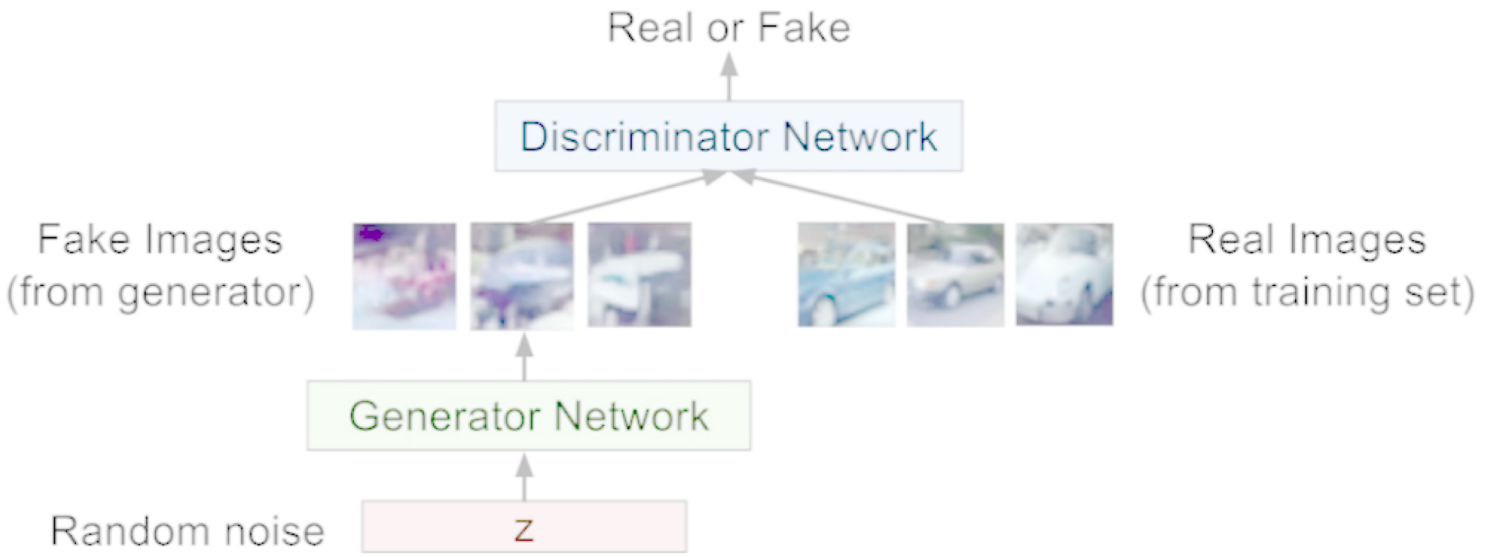
This network takes as input 100 random numbers drawn from a uniform distribution (we refer to these as a code, or latent variables, in red) and outputs an image.



https://blog.openai.com/generative-models/

Training GAN: two player game

- Generator network: try to foot the discriminator by generating real-looking images
- Discriminator network: try to distinguish between real and fake images



Training GAN: two player game

Train jointly minimax game.

Objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Discriminator output for real data x

Discriminator output for generated fake data G(z)

- Discriminator (θ_d) wants to **maximize objective** such that D(x) is close to 1 (real) and D(G(z)) is close to 0 (fake)
- Generator (θ_g) wants to **minimize objective** such that D(G(z)) is close to 1 (discriminator is fooled into thinking generated G(z) is real)

Training GANs

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

Alternate between:

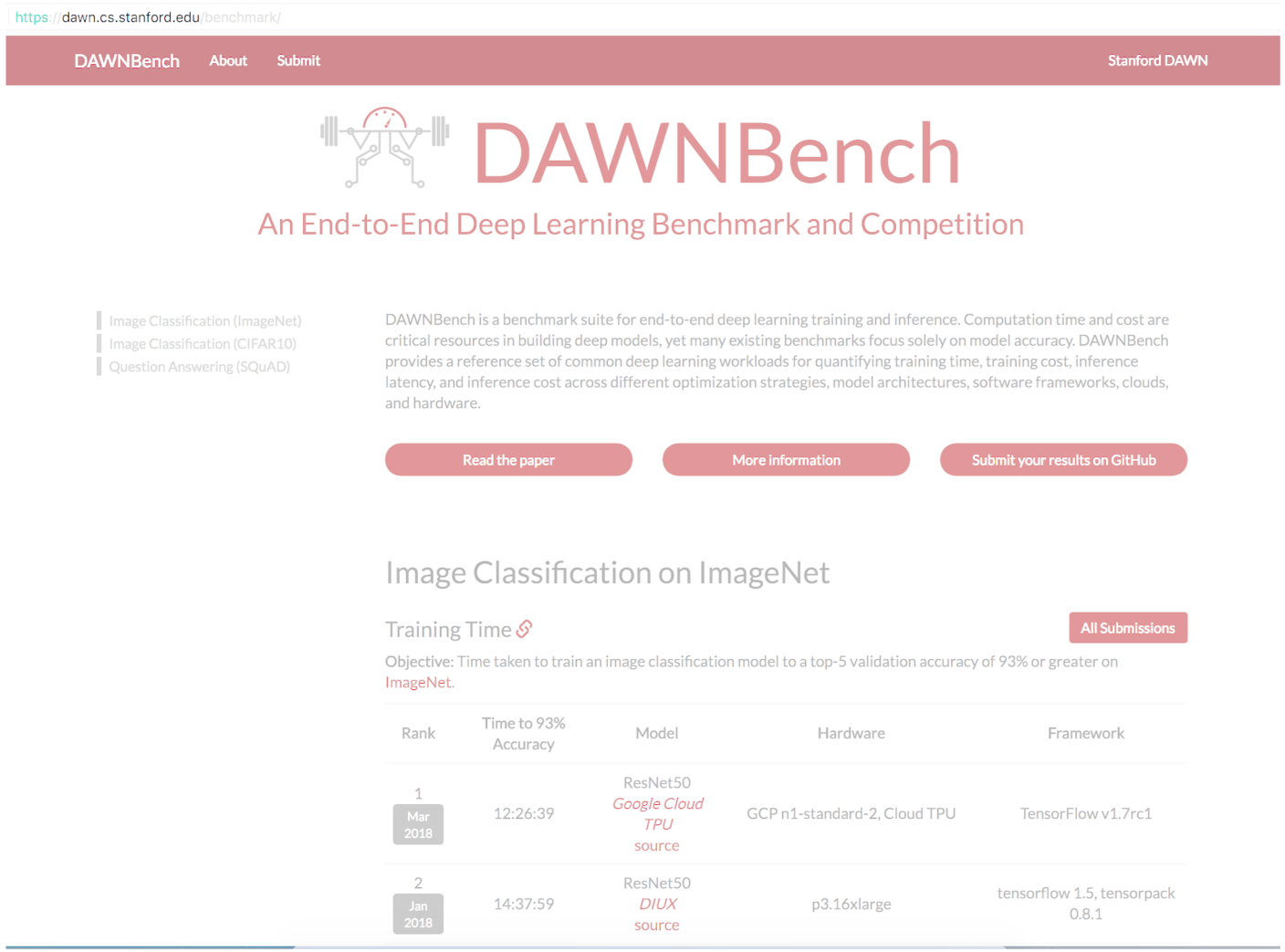
1. **Gradient ascent** on discriminator

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

2. **Gradient descent** on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Benchmark


https://dawn.cs.stanford.edu/benchmark/
DAWNBench
About
Submit
Stanford DAWN
DAWNBench
An End-to-End Deep Learning Benchmark and Competition
Image Classification (ImageNet)
Image Classification (CIFAR10)
Question Answering (SQuAD)
DAWNBench is a benchmark suite for end-to-end deep learning training and inference. Computation time and cost are critical resources in building deep models, yet many existing benchmarks focus solely on model accuracy. DAWNBench provides a reference set of common deep learning workloads for quantifying training time, training cost, inference latency, and inference cost across different optimization strategies, model architectures, software frameworks, clouds, and hardware.
Read the paper
More information
Submit your results on GitHub
Image Classification on ImageNet
Training Time
All Submissions
Objective: Time taken to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.
Rank
Time to 93% Accuracy
Model
Hardware
Framework
1
Mar 2018
12:26:39
ResNet50
Google Cloud TPU
source
GCP n1-standard-2, Cloud TPU
TensorFlow v1.7rc1
2
Jan 2018
14:37:59
ResNet50
DIUX
source
p3.16xlarge
tensorflow 1.5, tensorpack 0.8.1

GPU vs CPU

Type	Name	Clock Speed	Cores
CPU	Intel Xeon E7-2850	2.00Ghz	10
CPU	Intel BX80684I78700K	3.7 GHz	6
CPU	AMD EPYC 7601	3.2 GHz	32
GPU	Nvidia GEFORCE Titan X Pascal, 12 GB	1.5 GHz	3584
GPU	Nvidia GEFORCE GTX 1080 Ti (11GB)	1.5 GHz	3584
GPU	NVIDIA Tesla V100 GPUs 32 GB (Server)	7 TF	5120

TensorFlow

- Open source library for high performance numeric computation
  - First release 2015, Apache License 2017, Current stable version 1.6
- Processing is expressed in the form of computation graph
- Operates over tensors - n-dimensional arrays
- Computes the gradients using generalized back propagation algorithm
- TensorFlow can take advantage of multi core CPUs, GPUs and specialized TPU
- TensorFlow parts
  - Placeholders - takes input
  - Variables - mutable variables that TF optimizer can update
  - Layers
  - Cost measure
  - Pre-built optimizers

Why computation graph?

- Automatic differentiation
- Deployment to a python free server, phone
- Graph based optimization
- Compilation, kernel fusion, layout selection
- Automatic distribution to 100s of machines

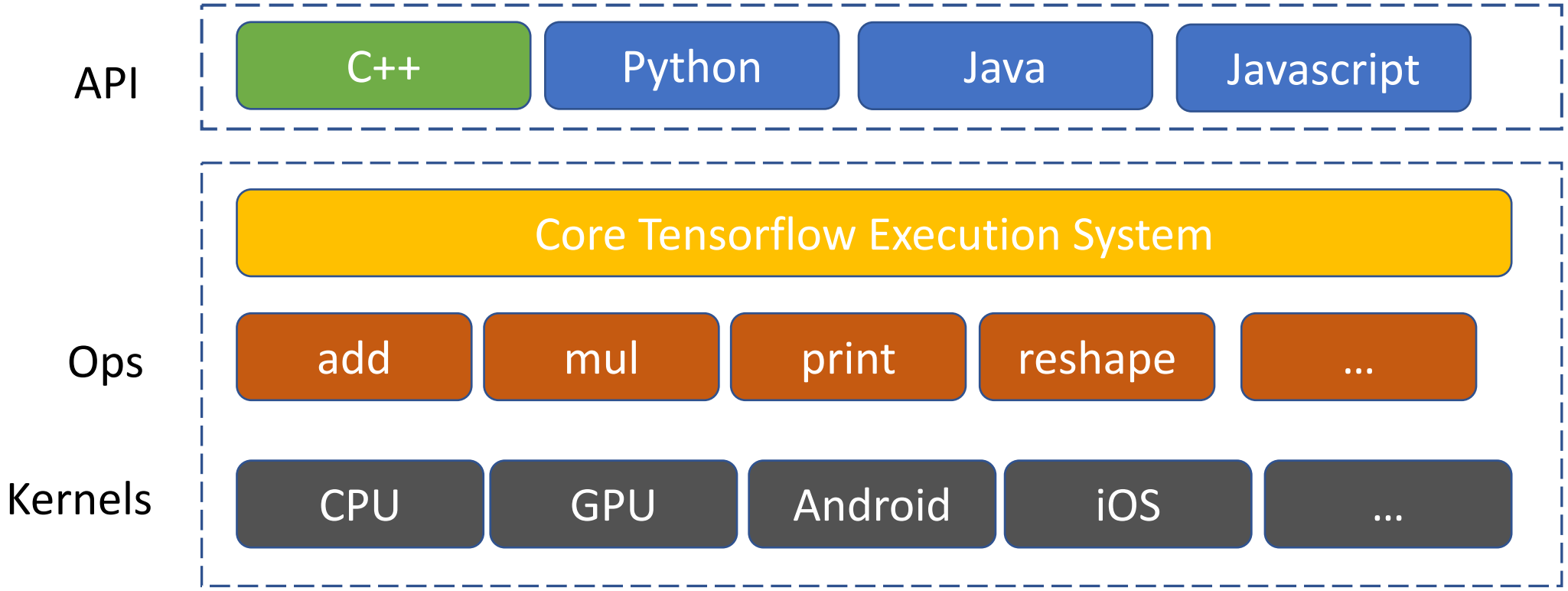
Lazy execution

- Lazy expressions are evaluated only when a dependent expression is evaluated. This evaluation strategy delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations.
- The benefits of lazy evaluation include:
  - The ability to define control flow (structures) as abstractions instead of primitives.
  - The ability to define potentially infinite data structures. This allows for more straightforward implementation of some algorithms.
  - Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions.
- Declarative programming style often uses lazy evaluation strategy.

Greedy/Eager execution model

- Eager evaluation is used by most traditional programming languages. In eager evaluation, an expression is evaluated as soon as it is bound to a variable.
- Advantages
  - Eliminates the need to track and schedule the evaluation of expressions.
  - Allows the programmer to dictate the order of execution, making it easier to control when sub-expressions (including functions) within the expression will be evaluated, as these sub-expressions may have side effects that will affect the evaluation of other expressions.
- Disadvantages
  - Forces the evaluation of expressions that may not be necessary at run time
  - May delay the evaluation of expressions that have a more immediate need

Tensorflow distributes computations across many machines, many types of machine



Tensorflow toolkit hierarchy

Estimator API	• High level out-of-box API compatible with scikit-learn
tf.layers, tf.losses, tf.metrics	• Components useful when building custom NN models
Core Tensorflow (Python)	• Python API gives full control
Core Tensorflow (C++)	• C++ API is quite low level
CPU, GPU,TPU, Android	• TF runs on different hardware

Tensorflow estimators

- Deep neural network

```
model = DNNRegressor(feature_columns=[...],
                     hidden_units=[128, 64, 32])
```

- Classification

```
model = LinearClassifier(feature_columns=[...])
model = DNNClassifier(feature_columns=[...], hidden_units=[...])
```

- Helper functions for other types of inputs

```
tf.contrib.layers.sparse_column_with_keys(column_name="gender",
                                          keys=["female", "male"])
```

Custom estimator