
“A beginning is the time for taking the most delicate care that the balances are correct.”
Frank Herbert, Dune
CMPUT 628
Deep RL
Marlos C. Machado
Class 3/ 25

Plan

Overview / Refresher of (everything?) Deep Learning

Warning! *This will be quick. It is meant to start establishing a common language between us, but it is too fast for you if you are seeing this for the first time.*

Reminder: You can still leave

- I know, I know, *Deep Reinforcement Learning* sounds fun, modern, and hyp-ey

But…

- But this course won't be so well-structured as you (or I) would hope
- I won't teach you how to code fancy deep RL algorithms
- I'm not as much fun as you might think
- I don't care about grades – I might have a reputation :-)
  - *There won't be a practice midterm*
- I don't care if this course ends up being difficult

Please, interrupt me at any time!

https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

MNIST and LeNet [Lecun et al.,1998]

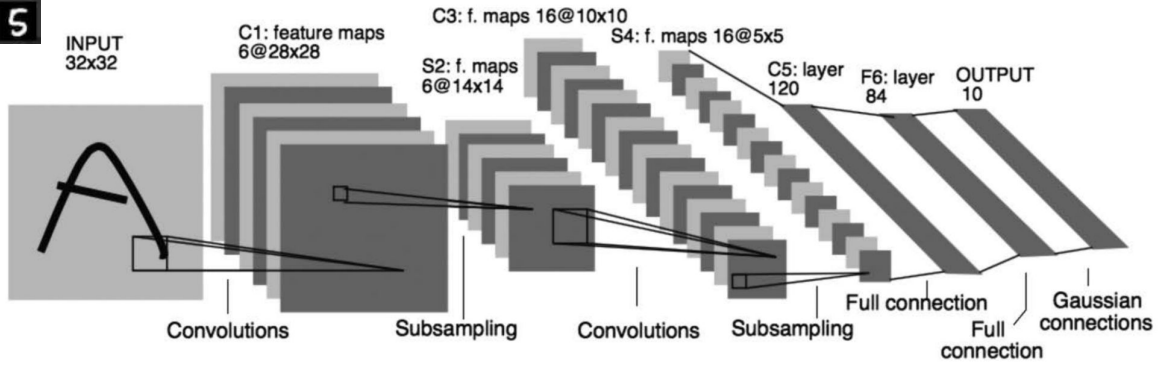
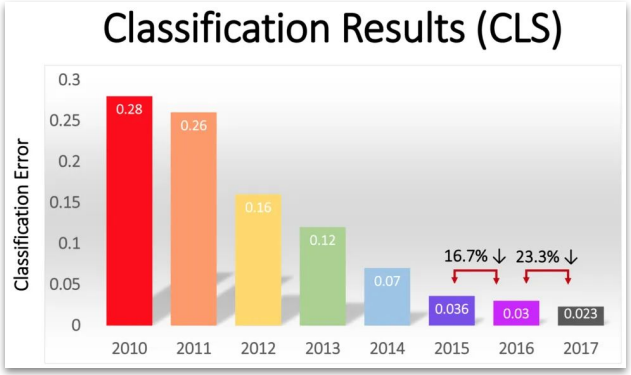
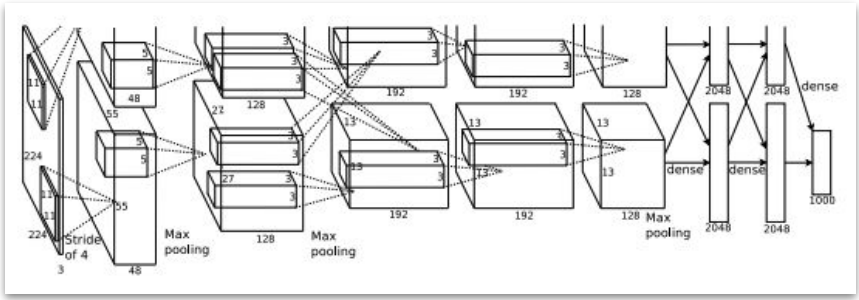


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

ImageNet



https://medium.com/@prudhvi.gnv/imagenet-challenge-advancement-in-deep-learning-and-computer-vision-124fd33cb948



Krizhevsky et al. (2012)

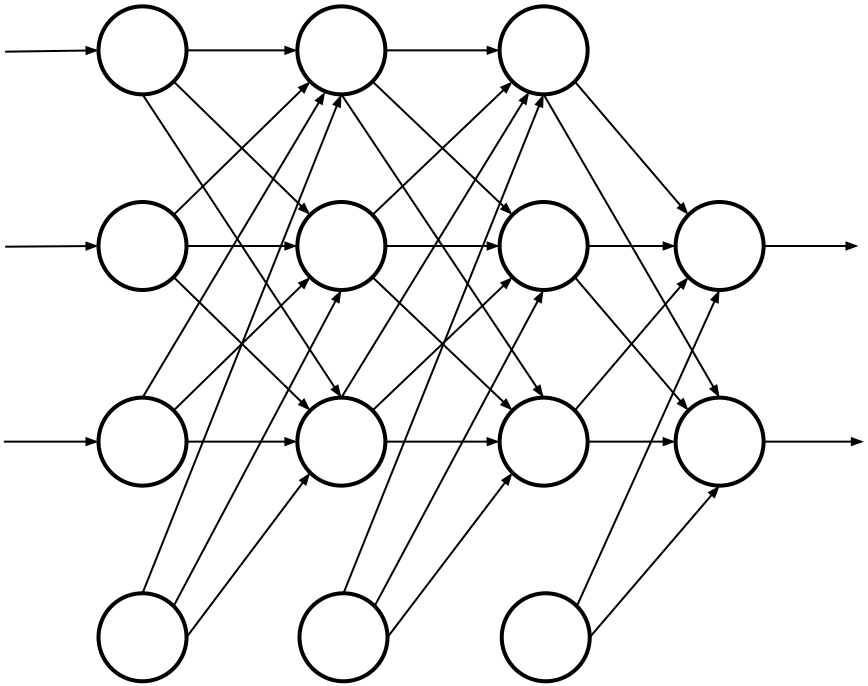
Nonlinear Function Approximation: Artificial Neural Networks

- The basics of deep reinforcement learning.
- Idea: Instead of using linear features, we feed the “raw” input to a neural network and ask it to predict the state (or state-action) value function.

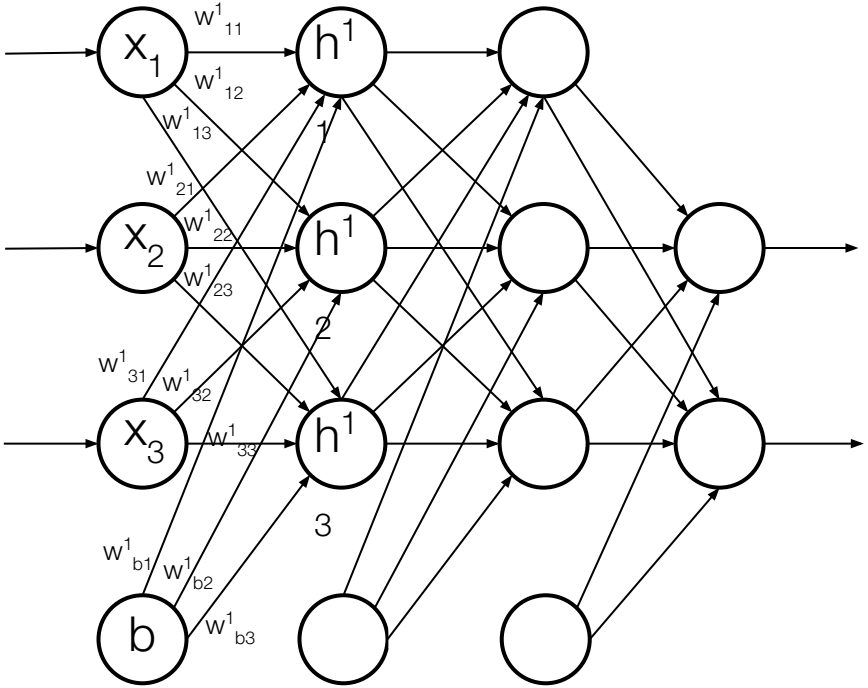
Neural Networks

Marlos C. Machado

Neural Networks



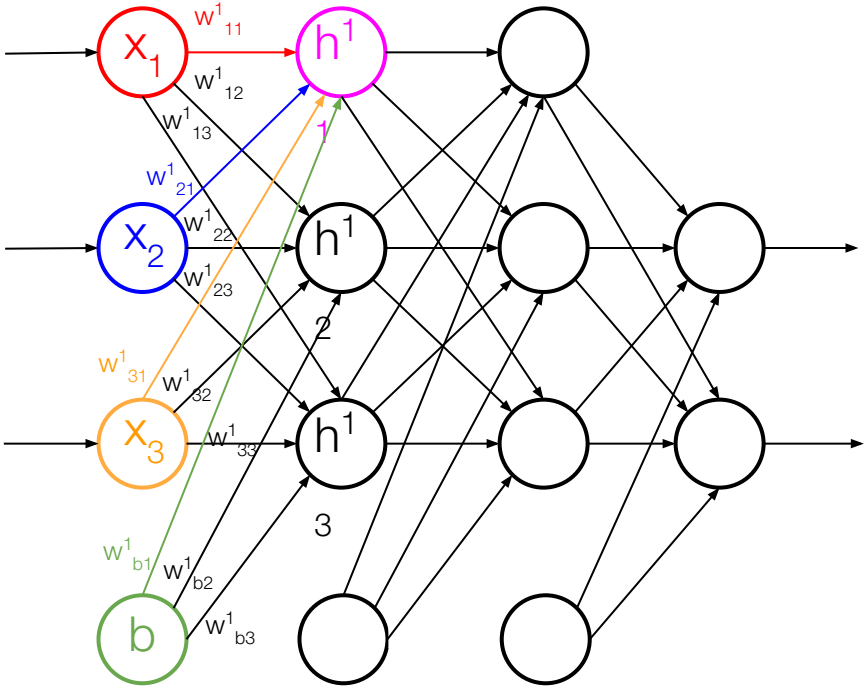
Neural Networks



$\mathbf{h^1} = \text{act}(\mathbf{xW^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

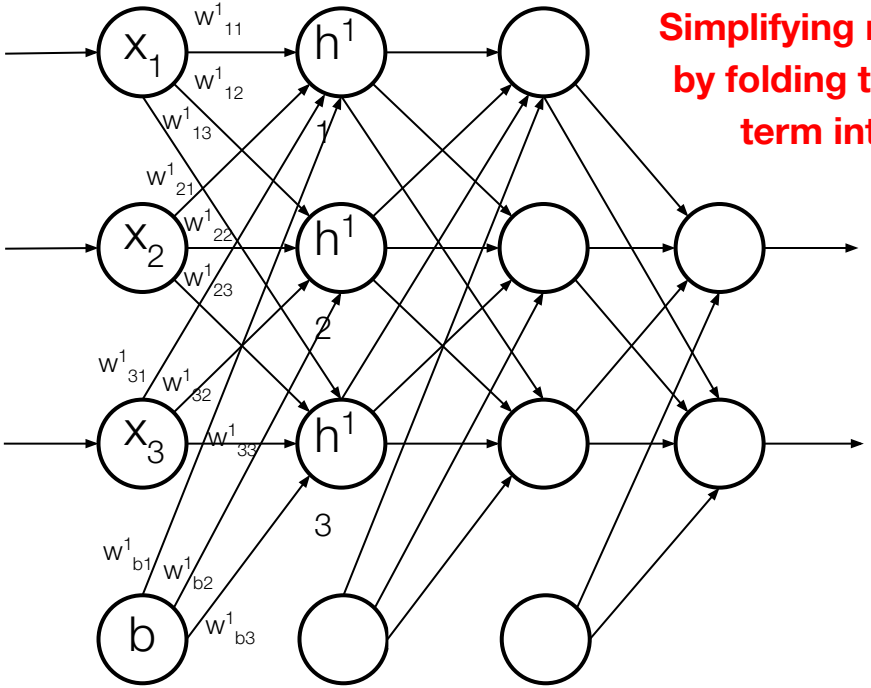
Neural Networks



$\mathbf{h^1} = \text{act}(\mathbf{x}\mathbf{W^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

Neural Networks



Simplifying notation by folding the bias term into x

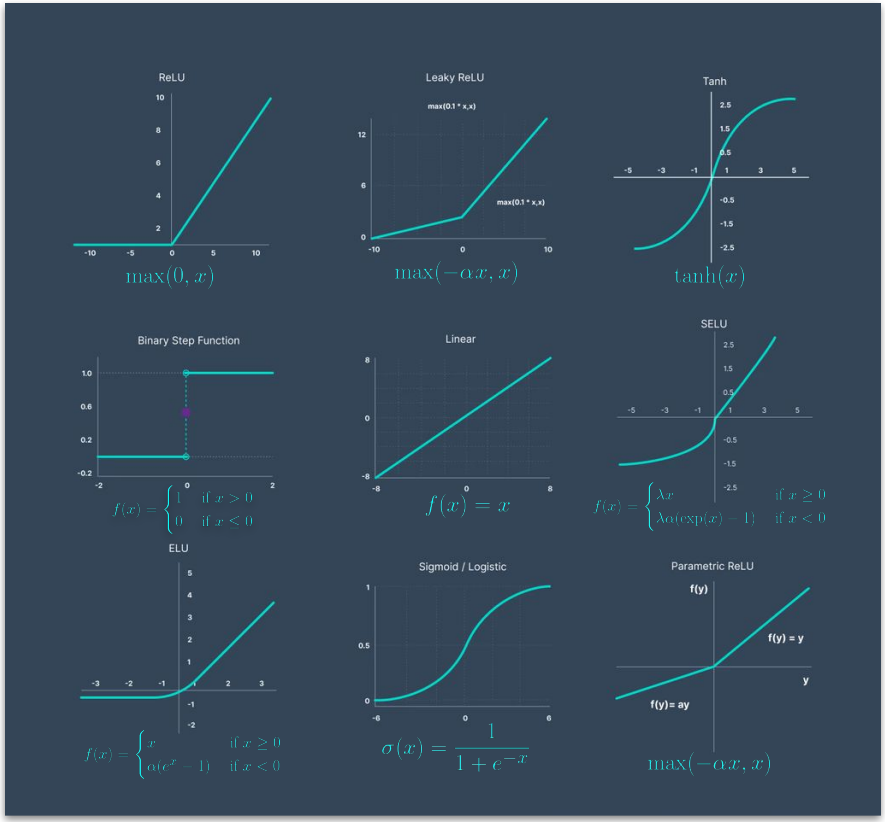
$\mathbf{h^1} = act(\mathbf{xW^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

The activation function introduces non-linearity

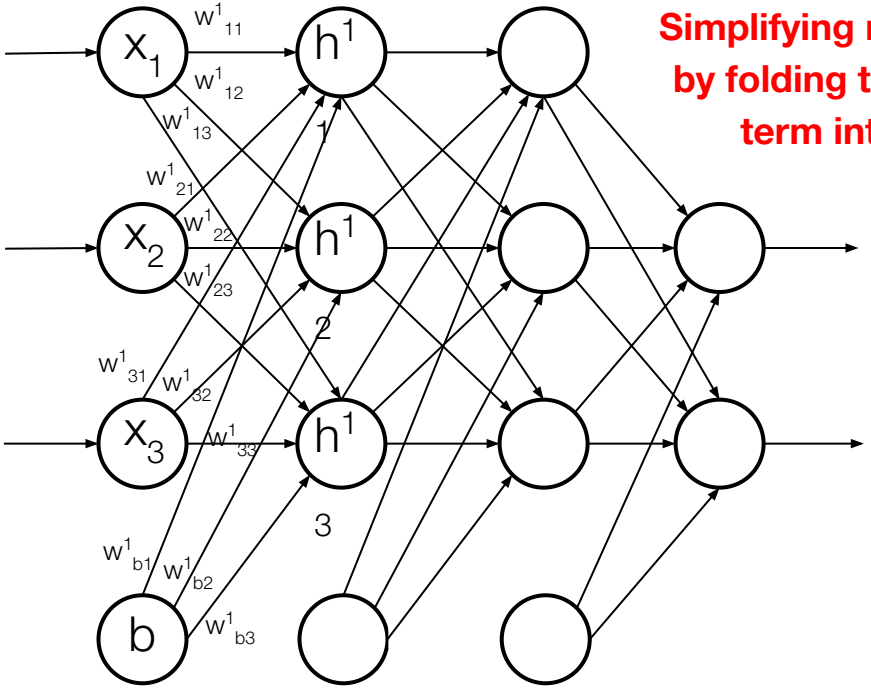
E.g.: *f(x) = max(0, x)*

Main Types of Activation Functions



https://www.v7labs.com/blog/neural-networks-activation-functions

Neural Networks



Simplifying notation by folding the bias term into x

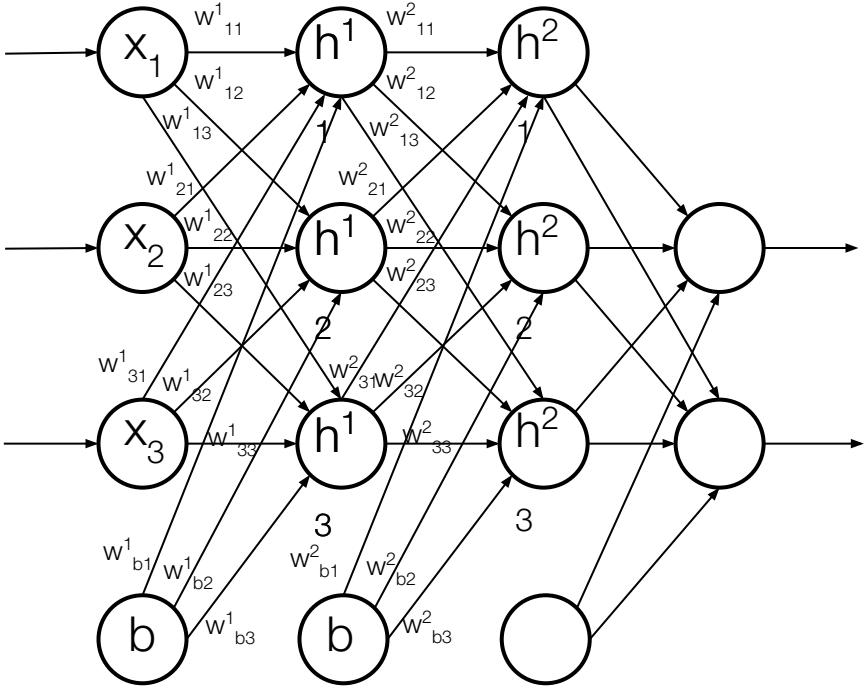
$\mathbf{h^1} = act(\mathbf{xW^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

The activation function introduces non-linearity

E.g.: *f(x) = max(0, x)*

Neural Networks



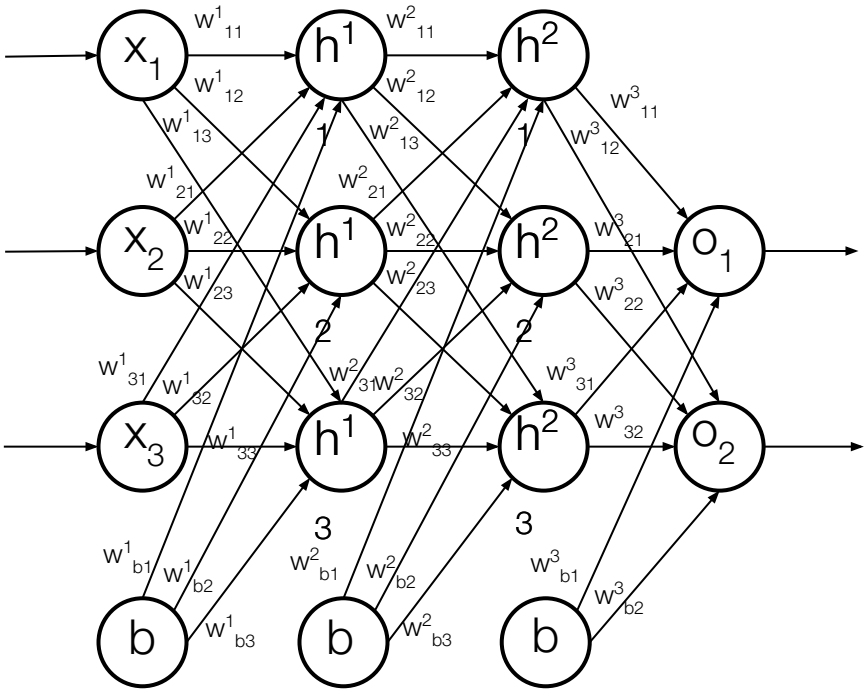
$\mathbf{h^1} = \text{act}(\mathbf{x}\mathbf{W^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

$\mathbf{h^2} = \text{act}(\mathbf{h^1}\mathbf{W^2})$

s.t. $h^2_1 = h^1_1 w^2_{11} + h^1_2 w^2_{21} + h^1_3 w^2_{31} + b w^2_{b1}$

Neural Networks



$\mathbf{h^1} = act(\mathbf{xW^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

$\mathbf{h^2} = act(\mathbf{h^1 W^2})$

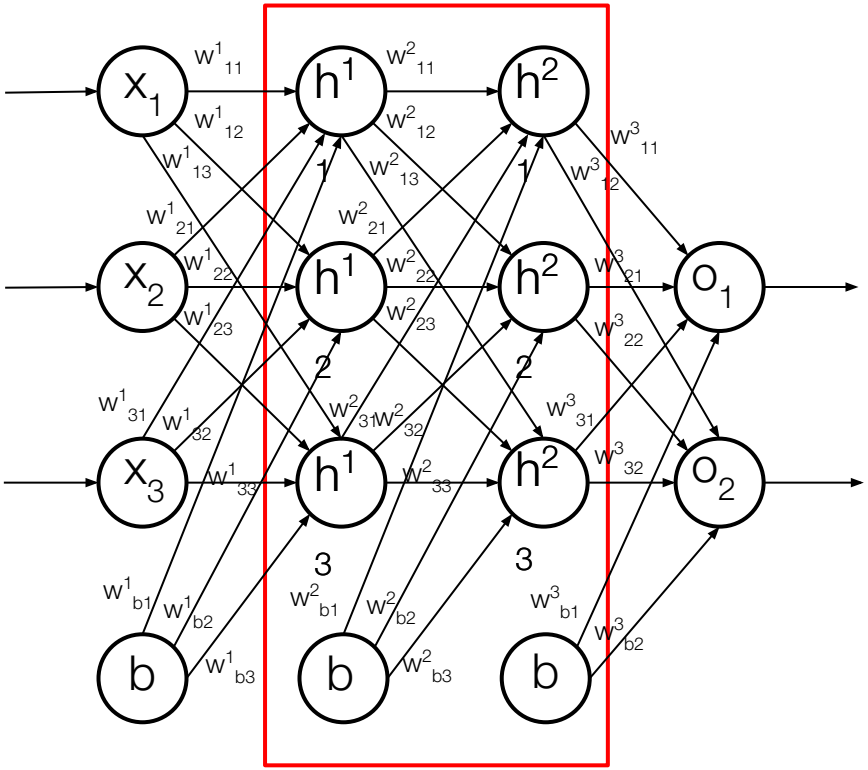
s.t. $h^2_1 = h^1_1 w^2_{11} + h^1_2 w^2_{21} + h^1_3 w^2_{31} + b w^2_{b1}$

$\mathbf{o} = act(\mathbf{h^2 W^3})$

s.t. $o_1 = h^2_1 w^3_{11} + h^2_2 w^3_{21} + h^2_3 w^3_{31} + b w^3_{b1}$

$\mathbf{o} = act(act(act(\mathbf{xW^1})\mathbf{W^2})\mathbf{W^3})$

Neural Networks



$\mathbf{h^1} = act(\mathbf{xW^1})$

s.t. $h^1_1 = x_1 w^1_{11} + x_2 w^1_{21} + x_3 w^1_{31} + b w^1_{b1}$

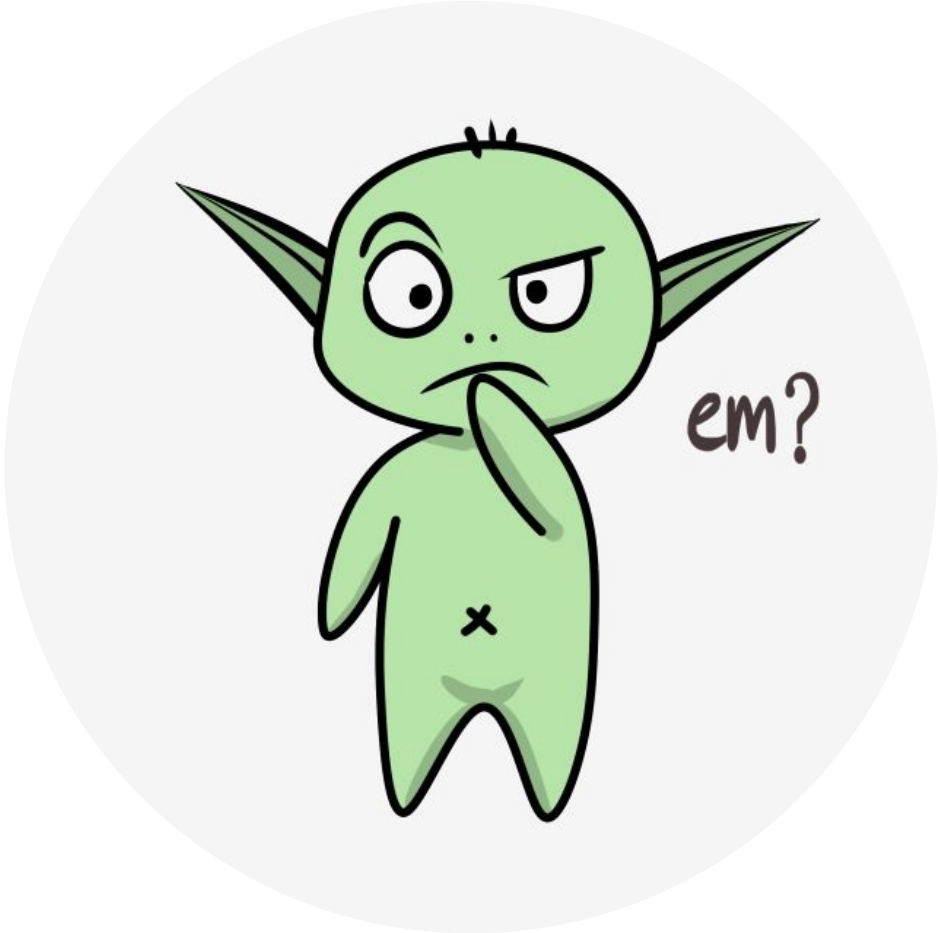
$\mathbf{h^2} = act(\mathbf{h^1 W^2})$

s.t. $h^2_1 = h^1_1 w^2_{11} + h^1_2 w^2_{21} + h^1_3 w^2_{31} + b w^2_{b1}$

$\mathbf{o} = act(\mathbf{h^2 W^3})$

s.t. $o_1 = h^2_1 w^3_{11} + h^2_2 w^3_{21} + h^2_3 w^3_{31} + b w^3_{b1}$

$\mathbf{o} = act(act(act(\mathbf{xW^1})\mathbf{W^2})\mathbf{W^3})$



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

How do we adjust the weights?

Stochastic Gradient Descent

- The nonlinearities make the problem non-convex
  (there's no algorithm with global convergence guarantees).
- The stochastic part of it allows us to consider one (or more) samples at a time, but it does not require us to process *all* samples before an update.
- We consider the impact that particular weight had in the final error we observed.
  - If we had no error, there's nothing to change :-)
  - If there was an error, we adjust the weights proportional to how big of a role they played.

But what error should we consider?

How do we figure out how big of a role a specific weight had?

But what error should we consider?
Examples of Loss Functions

Regression

Mini-batch

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Classification

True label

$$\text{H}(x) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{x} p(x_i)\log q(x_i)$$

Estimate

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \text{act}\left(\text{act}\left(\text{act}(\mathbf{x}\mathbf{W^1})\mathbf{W^2}\right)\mathbf{W^3}\right)\right)^2$$

⟨5, 4⟩, ⟨1, 0.5⟩, ⟨3, 2.7⟩

MSE = (1² + 0.5² + 0.3²)/3 = 1.34/3 = ~0.45

nple, 3 classes):

True: ⟨1, 0, 0⟩ Estimate ⟨0.7, 0.2, 0.1⟩

H(x) =~ 1 x 0.36 + 0 x 1.6 + 0 x 2.3 = ~0.36

How do we figure out how big of a role a specific weight had? Chain rule / Backpropagation [Rumelhart et al., 1986]

- To go from **x** to ŷ we do a forward pass, or forward propagation.
- Backpropagation (or backprop) allows the information to flow back from ŷ in order to compute the gradient.
- Backpropagation is not the whole learning algorithm, but just the method to compute the gradients.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\Big(y_i - \text{act}\big(\text{act}(\text{act}(\mathbf{x}\mathbf{W^1})\mathbf{W^2})\mathbf{W^3}\big)\Big)^2$$

$$\nabla_{\mathbf{x}} J(\mathbf{W})\,\mathbf{?}$$

Chain rule / Backpropagation [Rumelhart et al., 1986]

Explanation from Goodfellow, Bengio, and Courville (2016)

- Backpropagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.
- Suppose that y = g(x) and z = f(g(x)) = f(y). The chain rule states that $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$.
- Beyond the scalar case, let x $\in \mathbb{R}^m$, y $\in \mathbb{R}^n$, with g mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f mapping from $\mathbb{R}^n$ to $\mathbb{R}$. If **y** = g(**x**) and z = f(**y**), then $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$.
  In vector notation: $\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z$

  where ∂**y**/∂**x** is the n x m Jacobian matrix of g.
- The gradient of variable **x** can be obtained by multiplying a Jacobian matrix by a gradient. Backpropagation does so for each operation in the computation graph.

Chain rule / Backpropagation [Rumelhart et al., 1986]

Explanation from Goodfellow, Bengio, and Courville (2016)

- Suppose that x = f(w), y = f(x) and z = f(y), that is, z = f(f(f(w))).

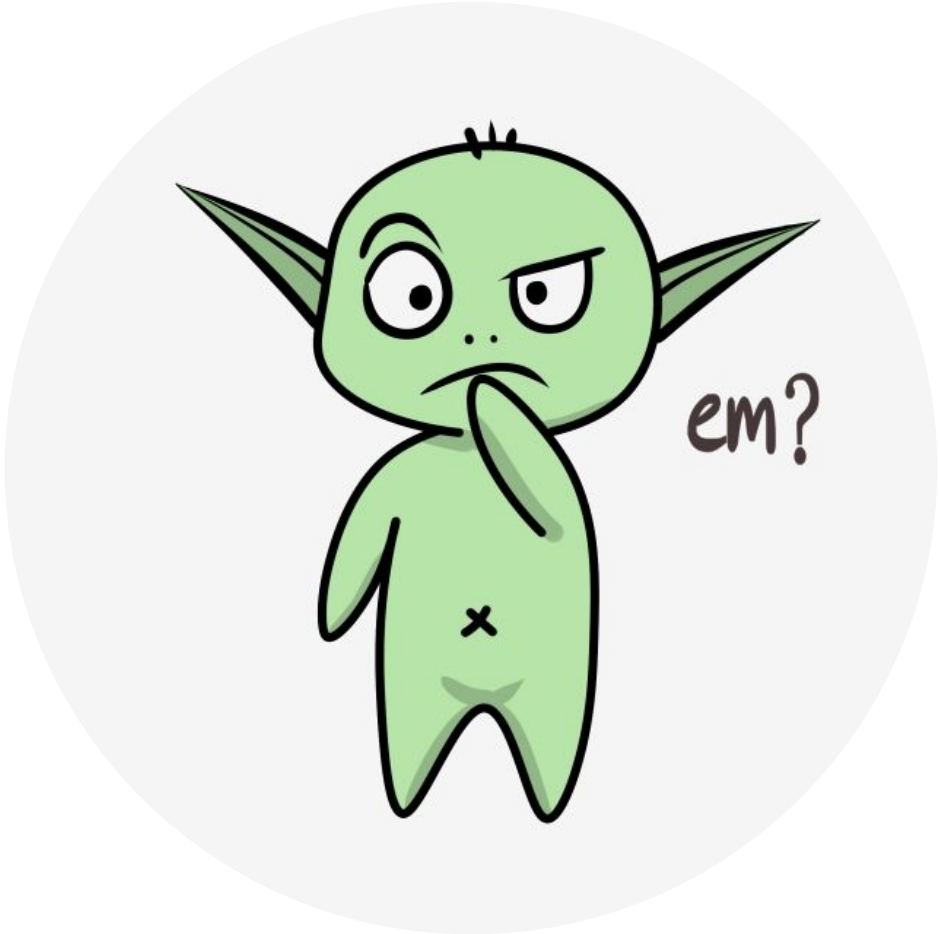
$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f'(y)f'(x)f'(w) = f'(f(f(w)))f'(f(w))f'(w).$$

Which one is better?

The first computes f(w) only once and store it in x. This is back-propagation.

When we don't have that much memory, we can use the second.

- "The back-propagation algorithm is designed to reduce the number of common subexpressions without regard to memory (...). Back-propagation thus avoids the exponential explosion in repeated subexpressions." [Goodfellow et al., 2016].



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

Optimization approaches

Stochastic gradient descent (SGD): $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha_k \, \hat{\boldsymbol{g}}$

- In practice, we need to decrease the step size* over time (thus the k) to reduce the noise introduced by the random sampling of minibatches
- Sometimes people use specific decay schemes: $\alpha_k \leftarrow (1 - \beta)\alpha_0 + \beta\alpha_\tau$
- There's no way around it, you need to tune the step size (and the other auxiliary variables) ¯_(ツ)_/¯
- Ideally, we would like a step size for each weight, and to adapt them automatically

* I suggest avoiding the expression *learning rate*

RMSProp [Hinton, 2012]

$$\boldsymbol{\nu} \leftarrow \rho\boldsymbol{\nu} + (1-\rho)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$$

$$\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\boldsymbol{\nu} + \epsilon}} \odot \hat{\boldsymbol{g}}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$

Exponentially weighted moving average accumulating the gradient

We individually scale the step-size by the size of the accumulated gradient. Weights with large partial derivatives have a rapid decrease in the step-size

This is what was first used by DQN. More on this later.

Adaptive Moments – Adam [Kingma and Ba, 2014]

$$\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1)\hat{\boldsymbol{g}}$$ ← Biased first moment estimate

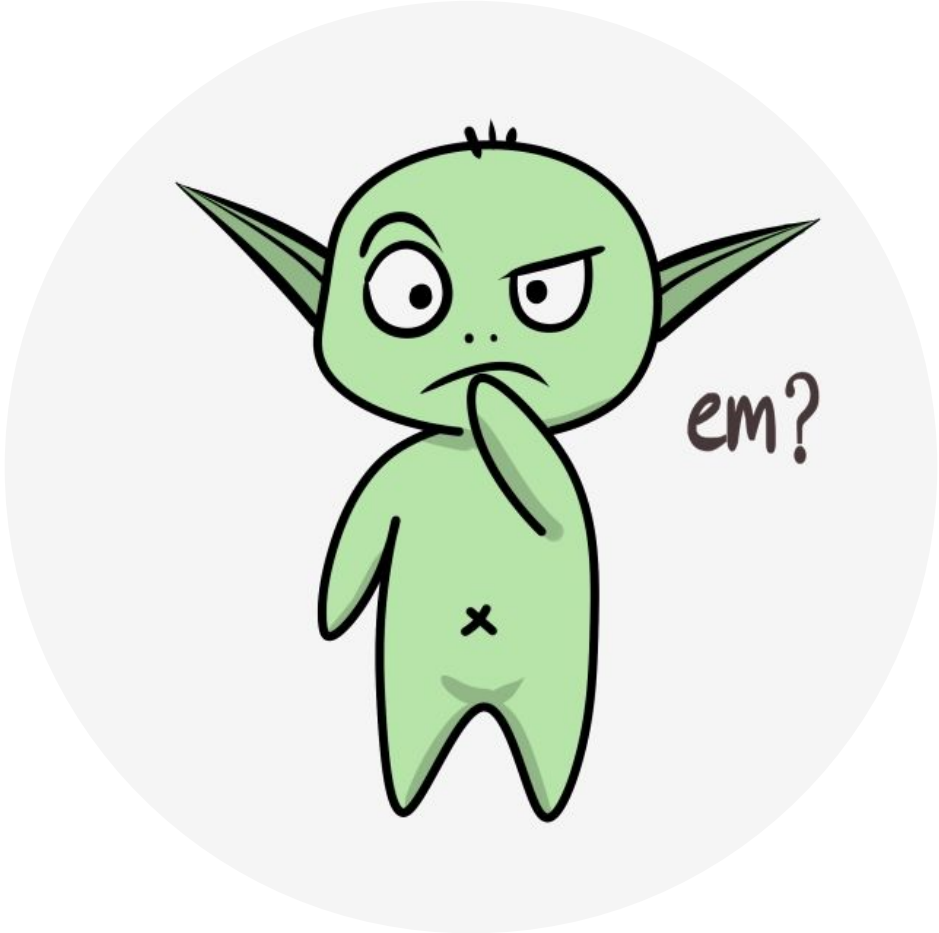
$$\boldsymbol{\nu} \leftarrow \beta_2 \boldsymbol{\nu} + (1 - \beta_2)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$$ ← Biased second moment estimate

$$\hat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1 - \beta_1^t}$$ ← Correct bias in first moment

$$\hat{\boldsymbol{\nu}} \leftarrow \frac{\boldsymbol{\nu}}{1 - \beta_2^t}$$ ← Correct bias in second moment

$$\Delta\boldsymbol{\theta} \leftarrow -\alpha \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{\nu}}} + \epsilon}$$ ← RMSProp-like update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$$



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

More Optimization for Deep Learning

Marlos C. Machado

Initialization *Really* Matters

- Many approaches try to ensure the weight matrix has nice properties before optimization begins

Uniform initialization

$$W_{i,j} \sim U\left(-\frac{1}{\sqrt{\text{fan}_{in}}}, \frac{1}{\sqrt{\text{fan}_{in}}}\right)$$

Xavier (or Glorot) initialization [Glorot and Bengio, 2010]

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}}, \sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}}\right)$$

He (or Kaiming) initialization [He et al,, 2015]

$$W_{i,j} \sim \mathcal{N}\left(0, {}^{2}/_{\text{fan}_{in}}\right)$$

Regularization: Parameter Norm Penalties

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

Generally we don't regularize the bias term

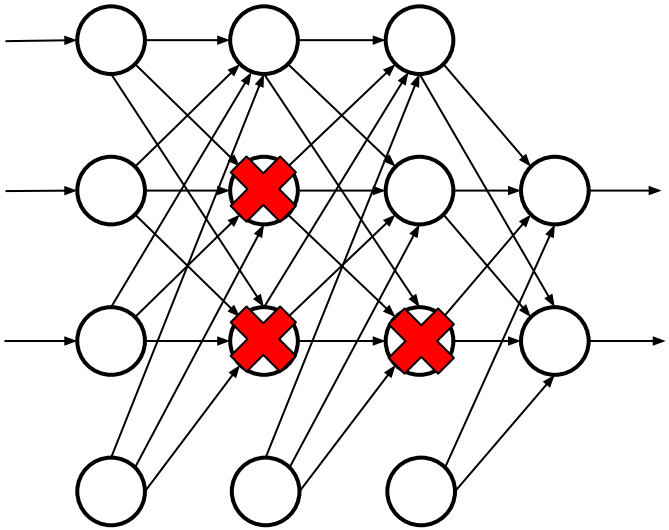
$$\Omega(\boldsymbol{\theta}) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 \qquad \Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|$$

ℓ_2, weight decay

ℓ_1, sparsity inducing

Regularization: Dropout [Srivastava et al., 2014]

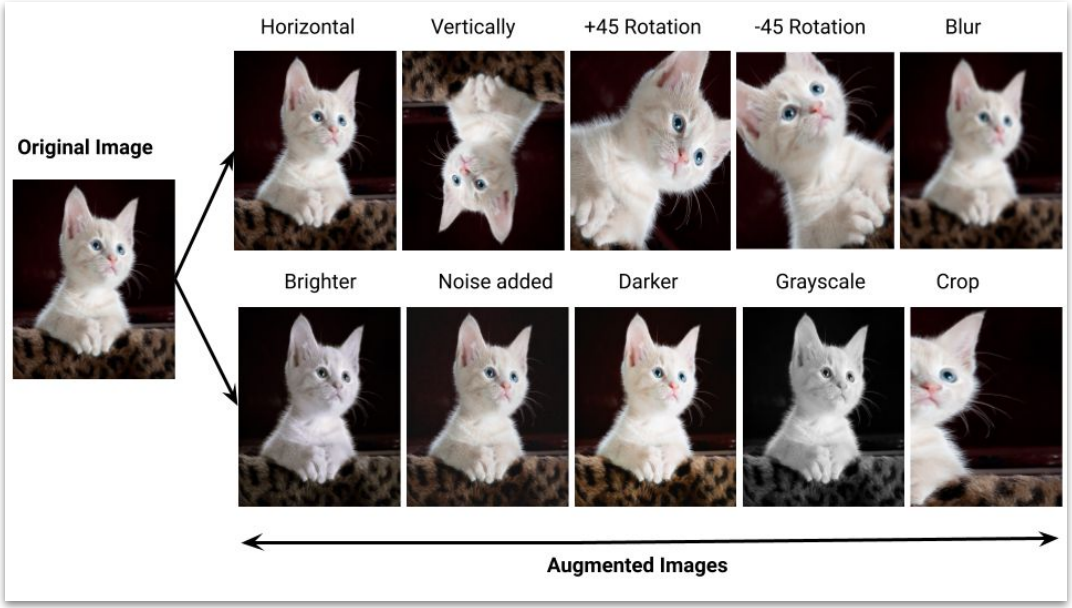
- Dropout is a cheap approximation for training multiple models at the same time.
- Dropout consists in training subnetworks obtained by removing nonoutput units from the original network. A unit needs to perform well regardless of the others.



- For each new minibatch, randomly sample a different binary mask.
  E.g., Hidden units are generally dropped w.p. 0.5.
- For inference, we multiply each weight by the probability of them not being dropped (weight scaling inference rule).
  E.g., Divide weights by 2 at the end of training.

Regularization: Data Augmentation

- How do we generate more data without actually having more data? Fake data!
  - Depending on the problem, it is relatively easy to generate fake data.



[Figure from https://ubiai.tools/what-are-the-advantages-anddisadvantages-of-data-augmentation-2023-update/]

Batch Normalization Layer [Ioffe and Szegedy, 2015]

- Instead of normalizing only the inputs, we normalize the input to every layer.
- "To shift and rescale each activation so that its mean and variance across the batch become values that are learned during training" [Prince, 2023].
  - Ensures variance is stable during forward pass at initialization.
  - Loss surface and its gradient change more smoothly, thus we can use larger step sizes.
  - Regularization. Batch normalization adds noise because the differences across batch statistics.

1 $$\hat{x}_i^{(k)} = \frac{x_i^k - \mu_B^{(k)}}{\sqrt{\left(\sigma_B^{(k)}\right)^2 + \epsilon}}$$

2 $$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

learnable parameters

3 We keep track of the exp. moving avg. so we can use that in inference (remember, we have one sample now)

Layer Normalization [Ba et al., 2016]

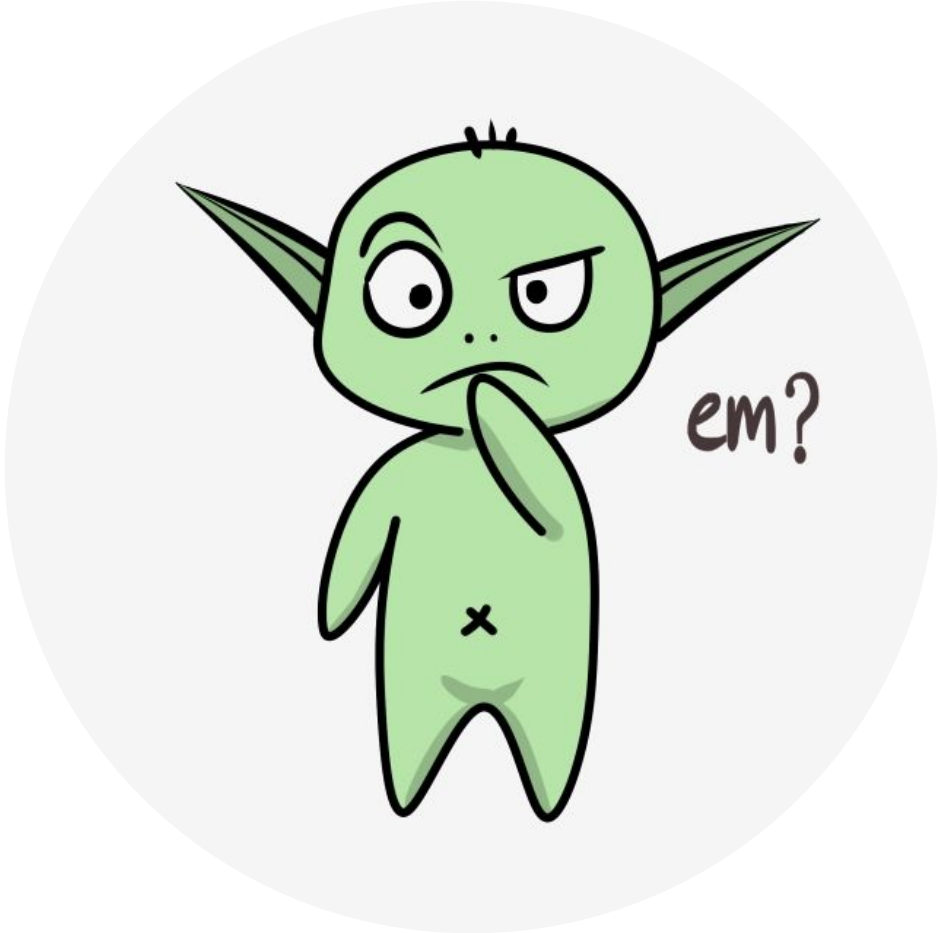
- Instead of normalizing individual features within a batch, we normalize the activations of each layer across the entire dataset.
  - Instead of computing the mean and std. dev., μ_B and σ_B, within a batch, we do so across all features (i.e. units), getting μ_L and σ_L.
  - Obviously, it has no dependency on the batch size. Thus, more amenable to sequences.
  - Similar to batch normalization, it can be applied before or after the activation function.

(1) $$\hat{x}_i^{(k)} = \frac{x_i^k - \mu_L^{(k)}}{\sqrt{\left(\sigma_L^{(k)}\right)^2 + \epsilon}}$$

(2) $$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

learnable parameters

(3) No need for special adjustments for inference.



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

Are we improving anything? From features to architecture…

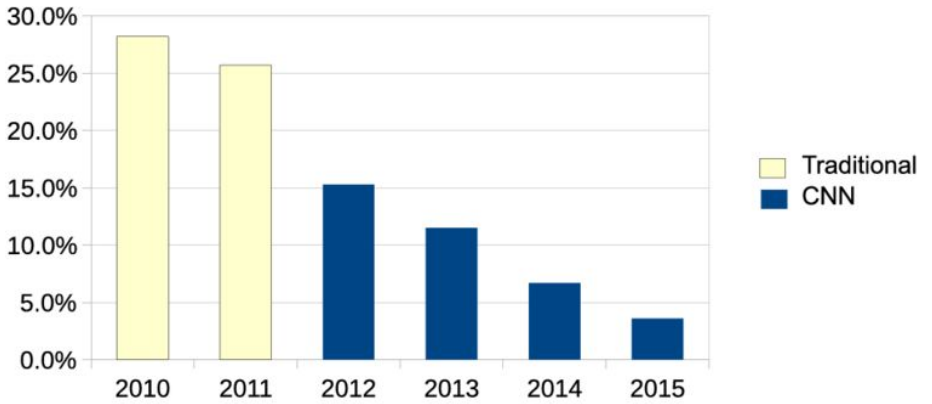
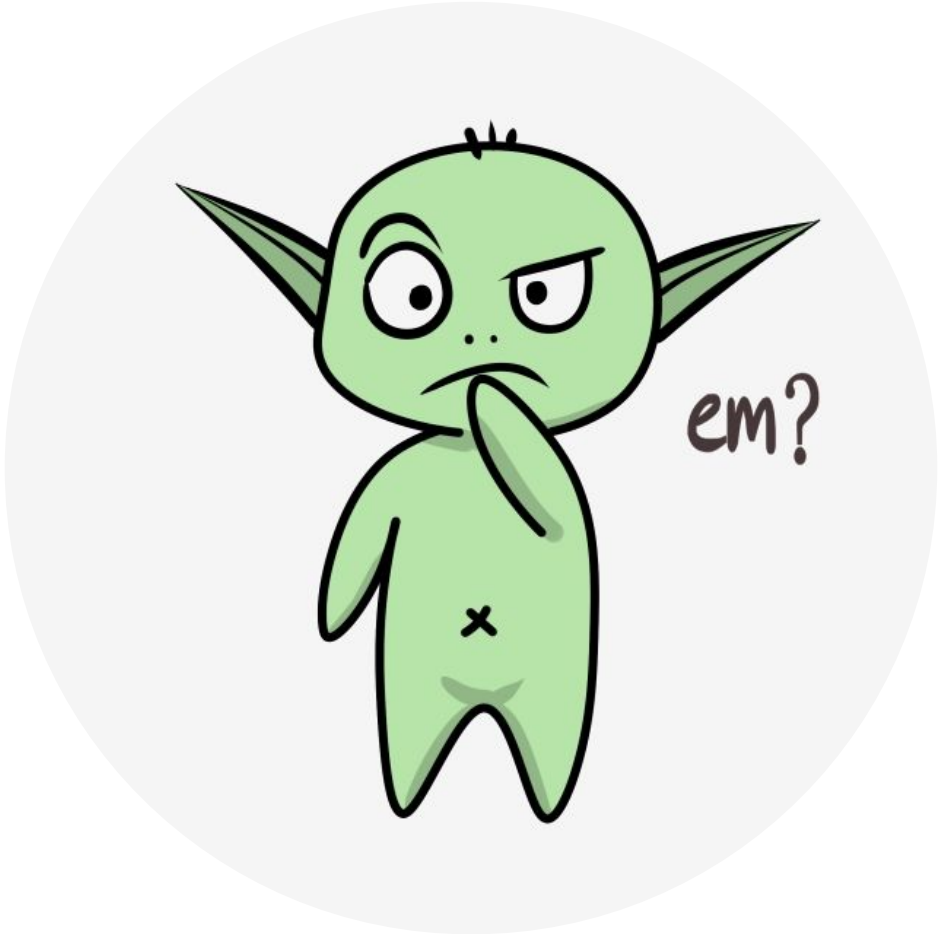


Image by Bottou, Curtis, and Nocedal (2016)



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

Convolutional Neural Networks

Marlos C. Machado

Deep Convolutional Network

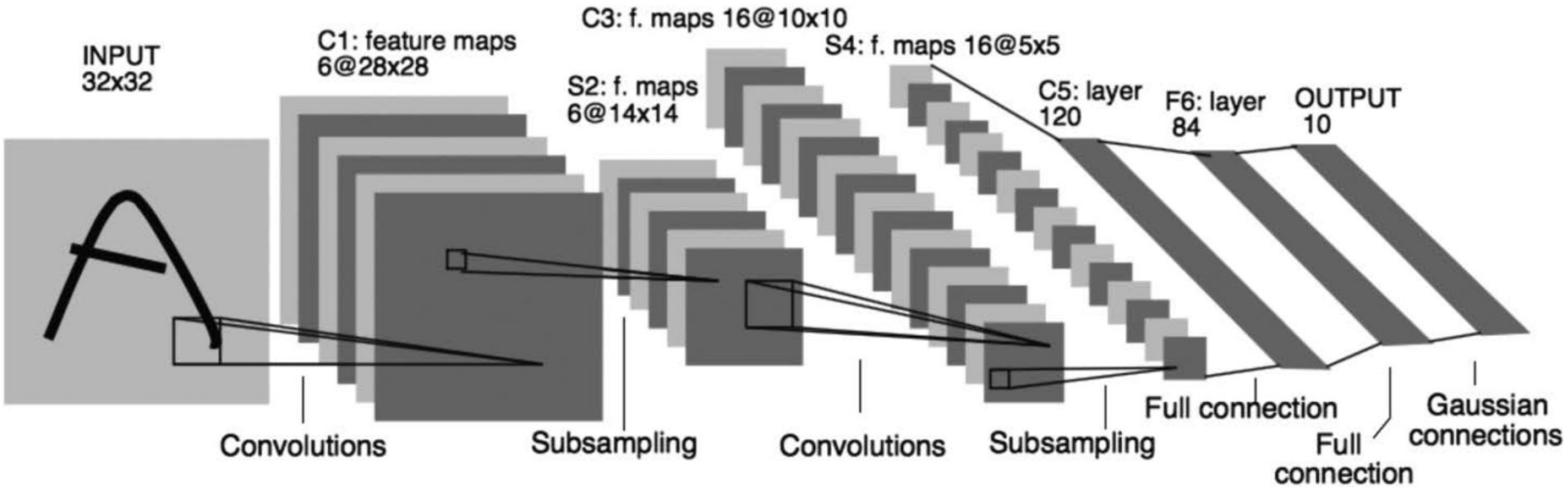
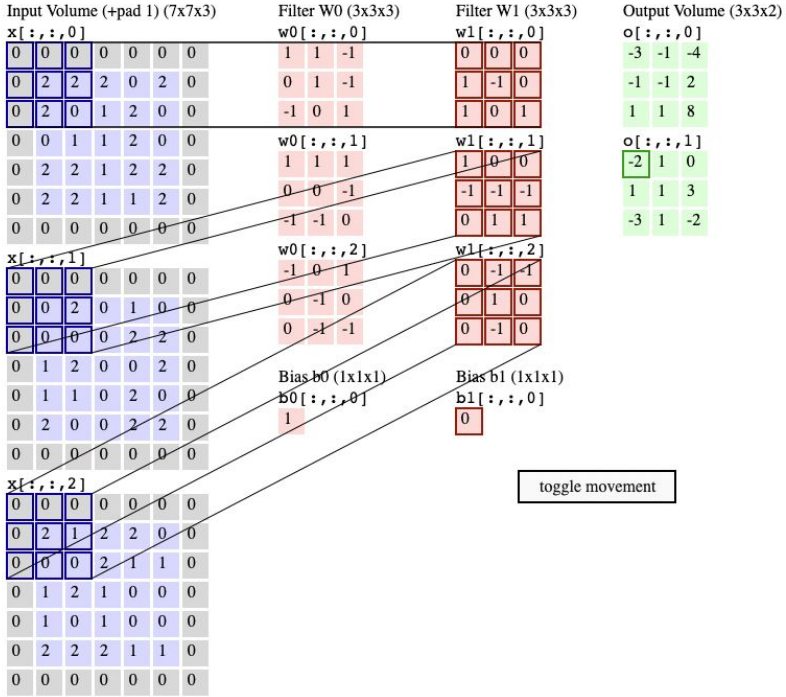


Figure 9.15: Deep Convolutional Network. Republished with permission of Proceedings of the IEEE, from Gradient-based learning applied to document recognition, LeCun, Bottou, Bengio, and Haffner, volume 86, 1998; permission conveyed through Copyright Clearance Center, Inc.

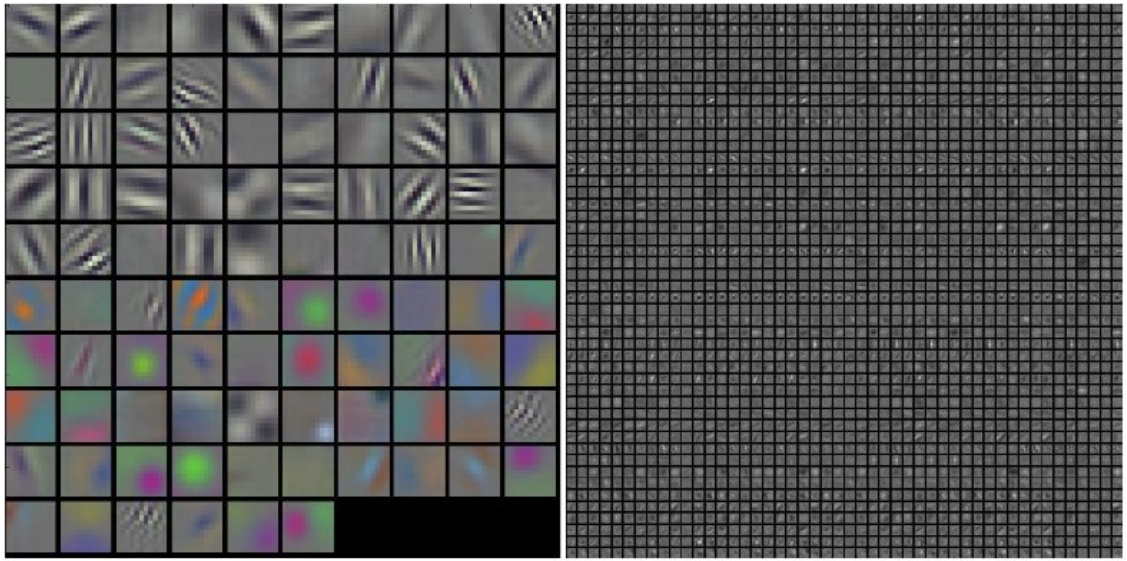
Deep Convolutional Network

Notice weights are shared



[Figure from demo in https://cs231n.github.io/convolutional-networks/]

Learned Representations



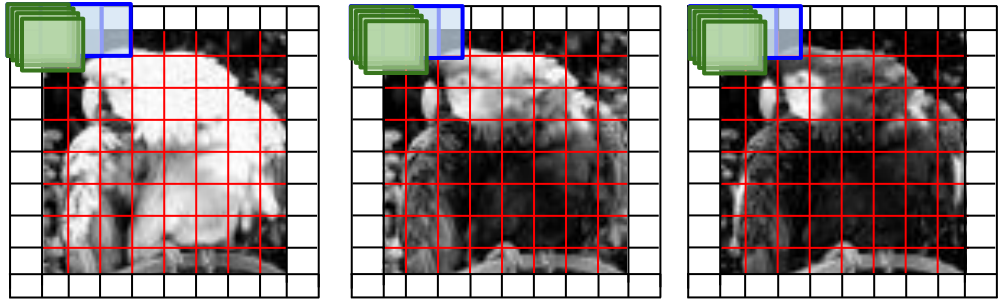
Typical-looking filters on the first CONV layer (left), and the 2nd CONV layer (right) of a trained AlexNet. Notice that the first-layer weights are very nice and smooth, indicating nicely converged network. The color/grayscale features are clustered because the AlexNet contains two separate streams of processing, and an apparent consequence of this architecture is that one stream develops high-frequency grayscale features and the other low-frequency color features. The 2nd CONV layer weights are not as interpretable, but it is apparent that they are still smooth, well-formed, and absent of noisy patterns.

[Figure from https://cs231n.github.io/understanding-cnn/]

The hyperparameters of a convolutional neural network

8x8x3 image **num. filters=4**

filter size=(2,2)



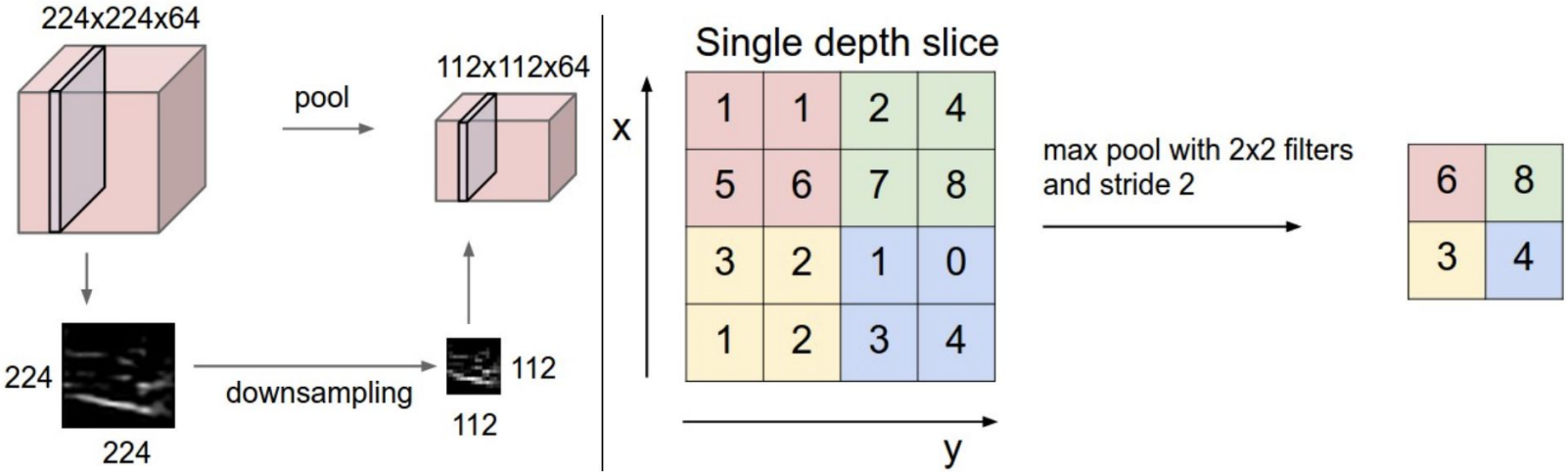
stride=2

padding=1
image
becomes
10x10x3

Size of output: (W - F + 2P)/S + 1

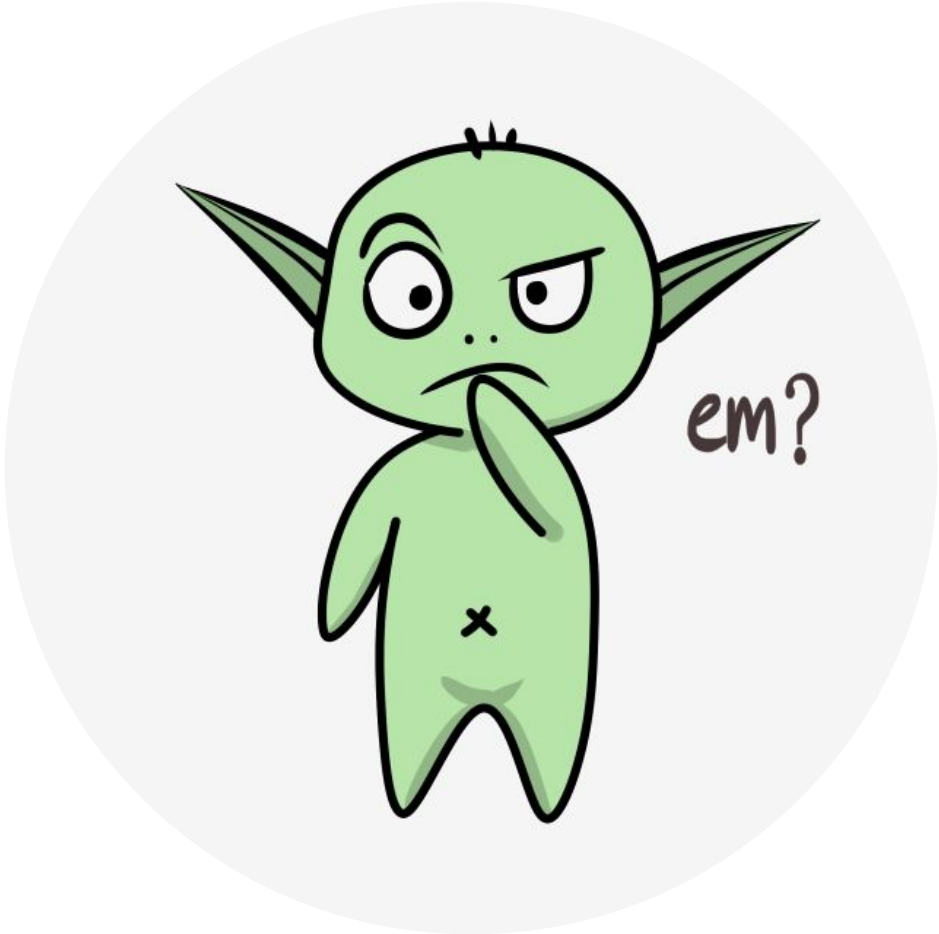
[https://upload.wikimedia.org/wikipedia/commons/5/56/RGB_channels_separation.png]

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).

[Figure from https://cs231n.github.io/understanding-cnn/]



https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

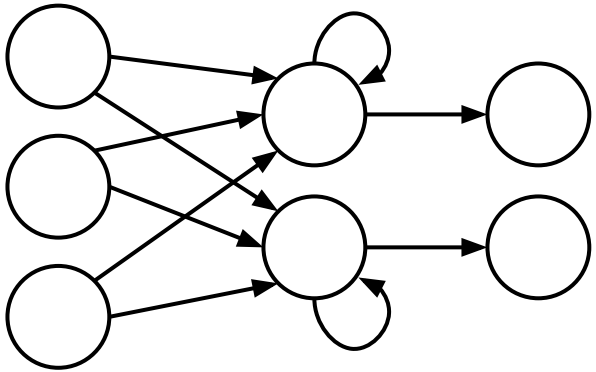
Recurrent Neural Networks

Marlos C. Machado

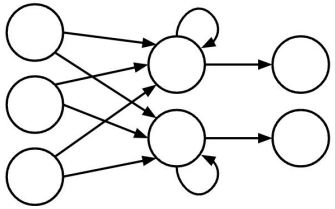
Recurrent Neural Networks (RNNs)

- RNNs are generally used to process *sequential* data.
- Similarly to how convolutional networks share parameters across patches of an image, RNNs share parameters across positions in a sequence.

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$



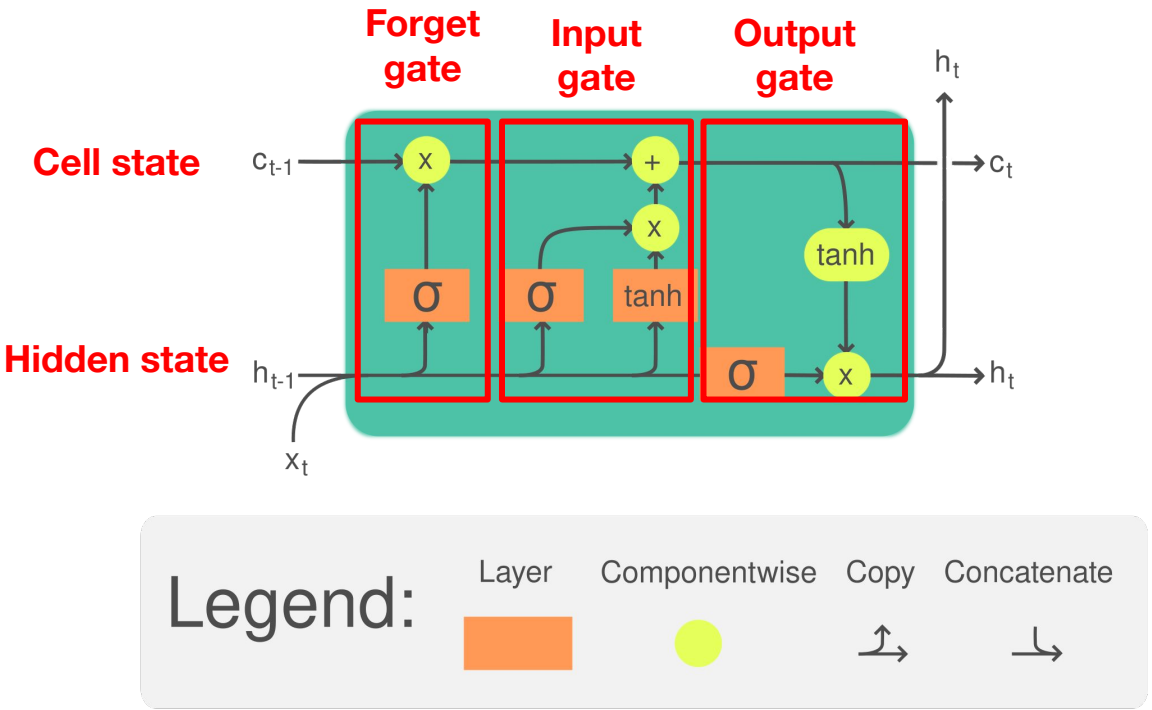
Recurrent Neural Networks (RNNs)



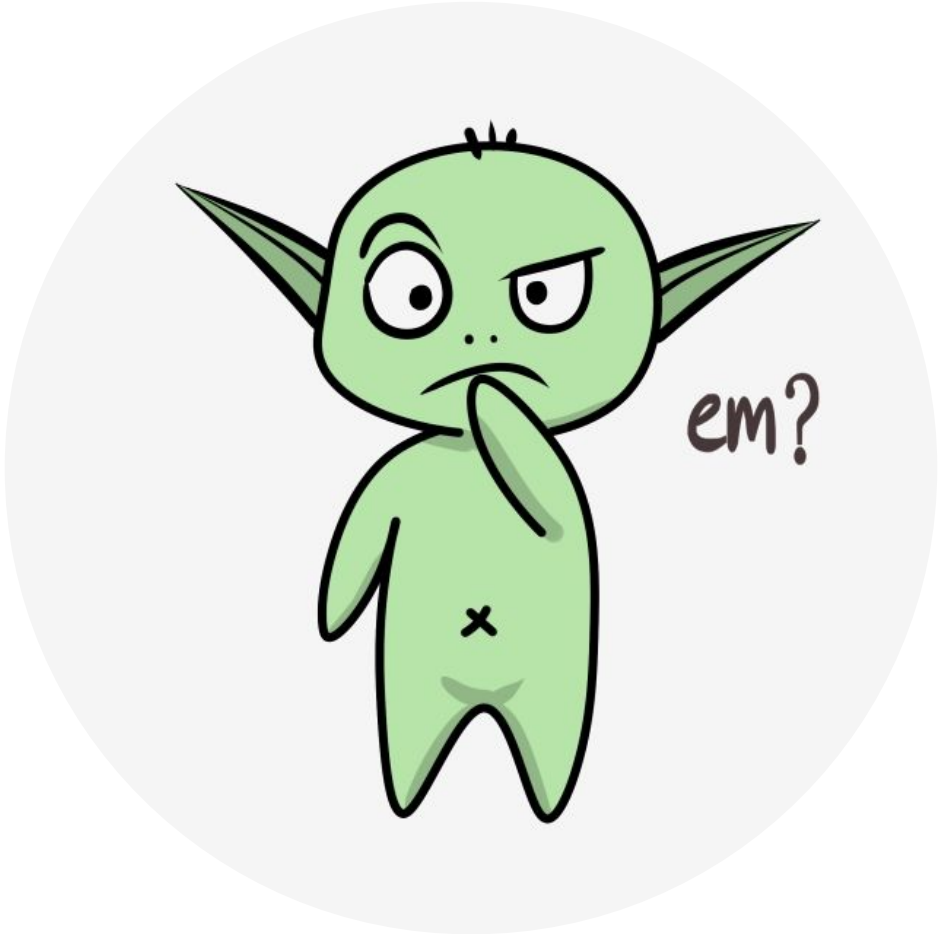
- RNNs are often trained using backpropagation through time, which like regular backpropagation but applied to an *unrolled* computational graph.
- They need to be trained in sequence.
- Their recurrence potentially gives us a lossy *memory*.
- In reinforcement learning, they are a natural instantiation of an agent state (not to be confused with the environment state nor the observation).

$$S_{t+1} \doteq u(S_t, A_t, O_{t+1})$$

Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997]



[Image from https://en.wikipedia.org/wiki/Long_short-term_memory]

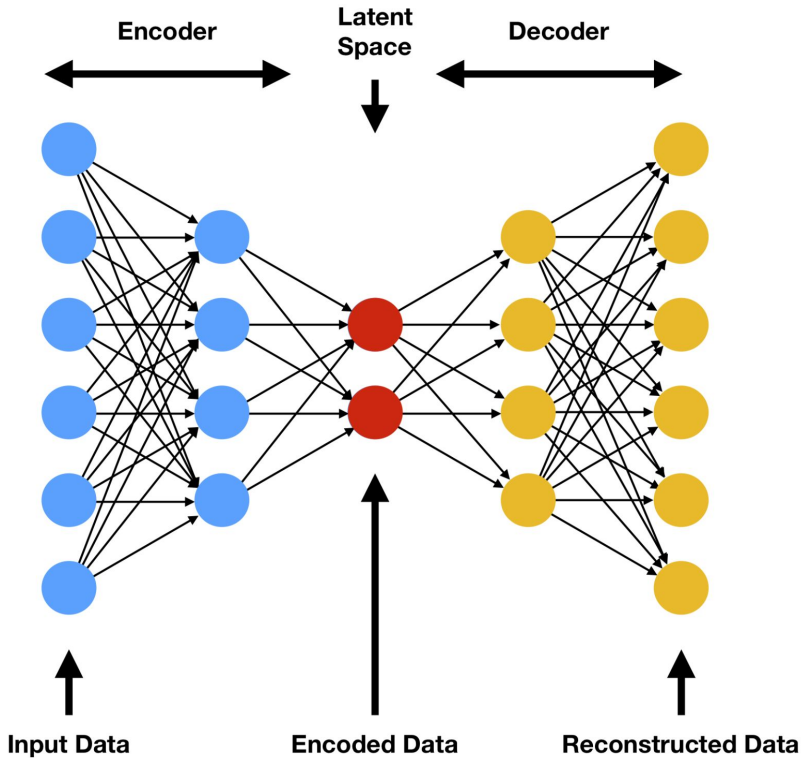


https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

Other Neural Network Architectures

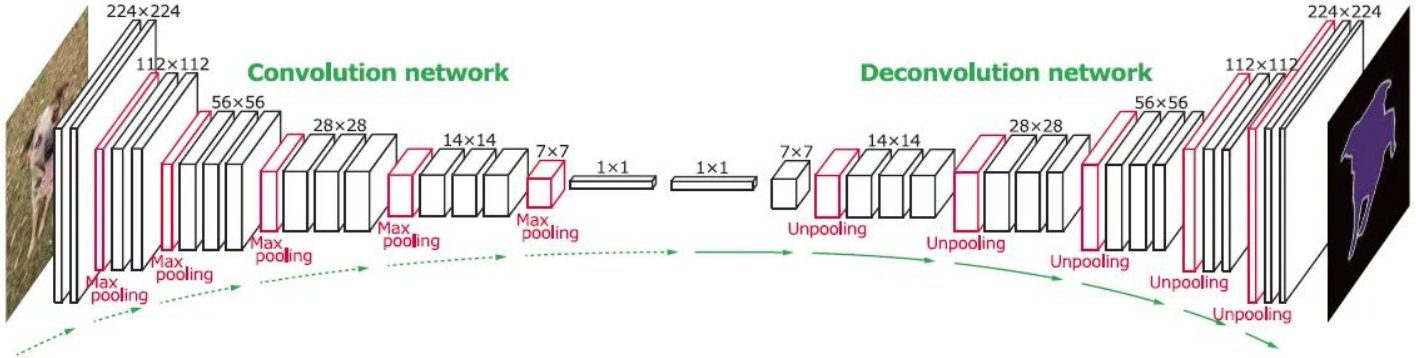
Marlos C. Machado

Autoencoders



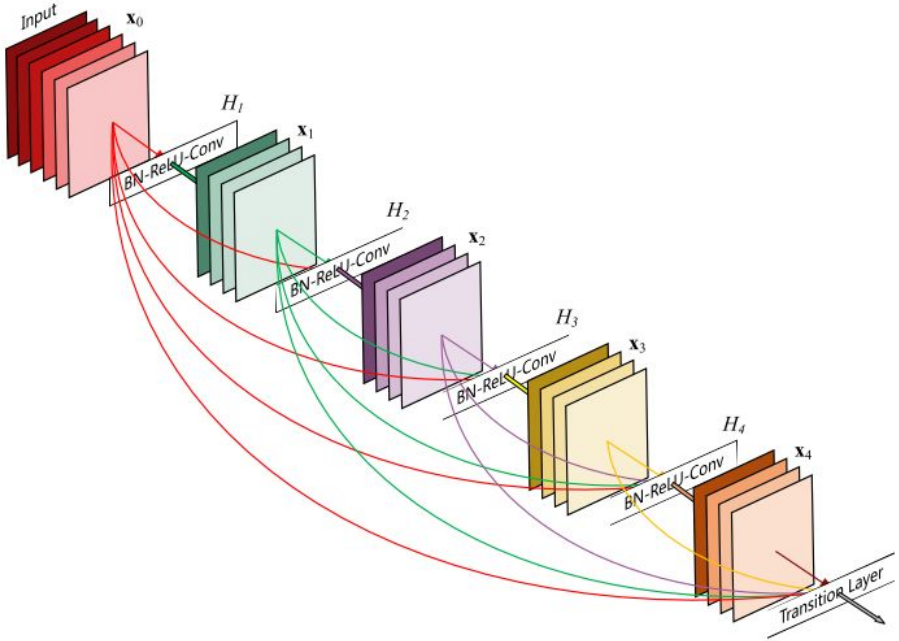
[Image from https://www.compthree.com/blog/autoencoder/]

Inverse Convolutional Network



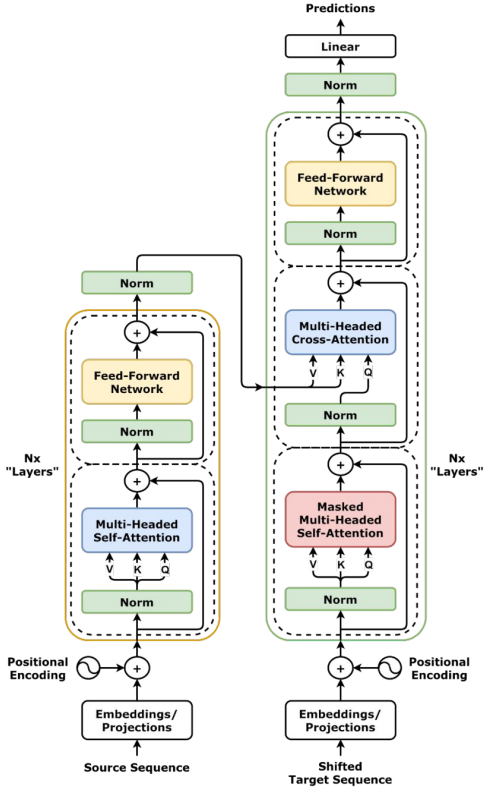
[Image from https://towardsdatascience.com/review-deconvnet-unpooling-layer-semantic-segmentation-55cf8a6e380e]

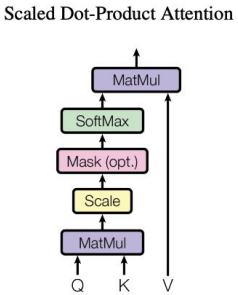
Residual Networks

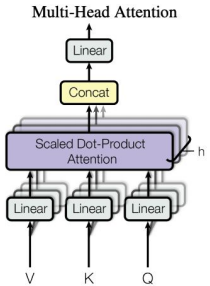


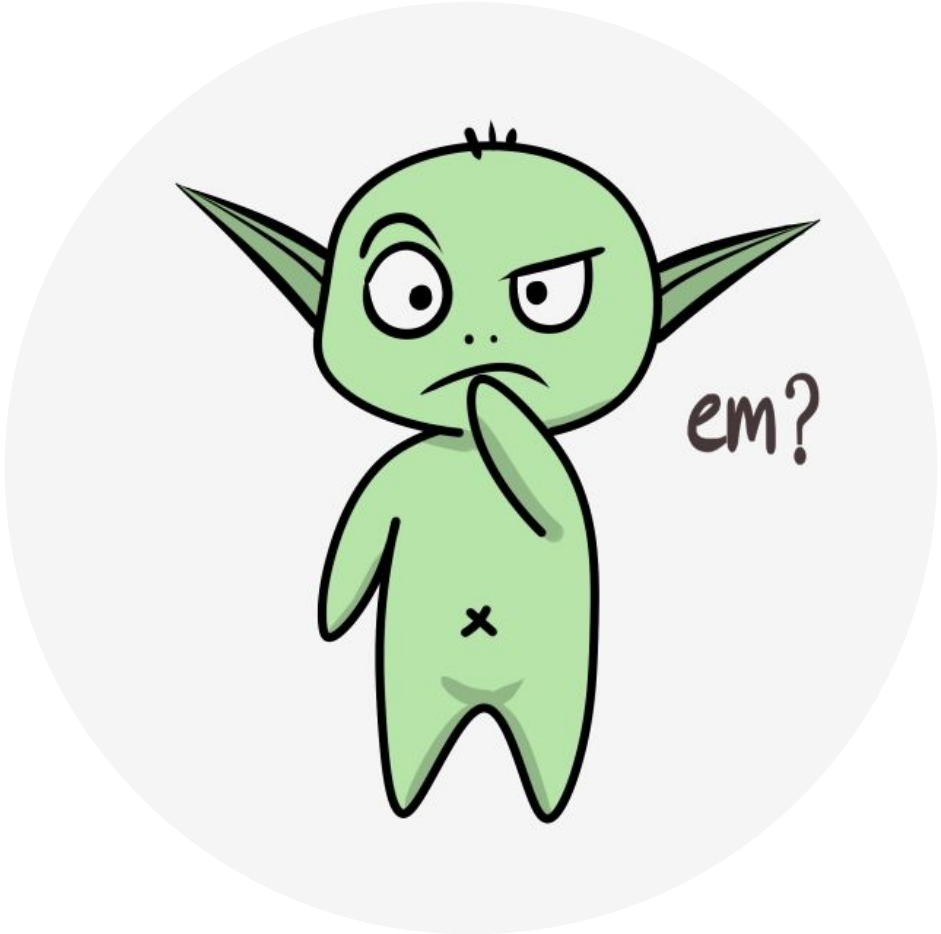
[Image from https://stats.stackexchange.com/questions/558130/are-there-any-weight-matrices-of-residual-connections-in-resnet]

Transformers[Vaswani et al., 2017]









https://pngtree.com/freepng/question-expression-cartoon-illustration_4545209.ht

Next class

- What I plan to do:
  - Finish overview of neural networks / deep learning.
  - Start talking about Deep RL; more specifically, DQN.
  - I'll try to make more lecture notes available.

- What I recommend YOU to do for next class:
  - *Brush-up* on the basics of deep learning if you don't remember.

    Specifically, Goodfellow, Bengio & Courville (2016)'s chapters 6–10.
  - Read the DQN paper: *V. Mnih et al.: Human-level control through deep reinforcement learning. Nature 518(7540): 529-533 (2015).*