Introduction to Deep Learning

Karthik Kashinath
NERSC
Created by: Mustafa Mustafa

AI4ESS Summer School
23, June 2020

Deep Learning -- Success stories

Dexterity, OpenAI, 2019

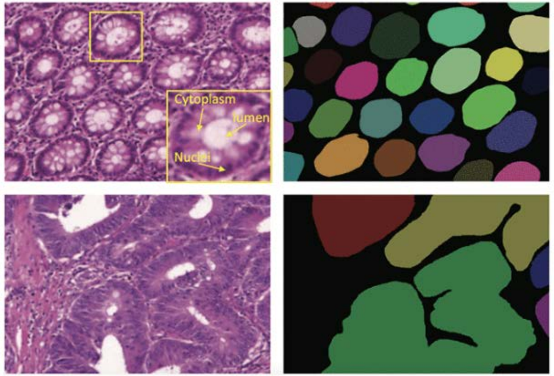
GANs Face Generation, becominghuman.ai, 2019

Deep Learning -- Success stories

Self-driving Cars

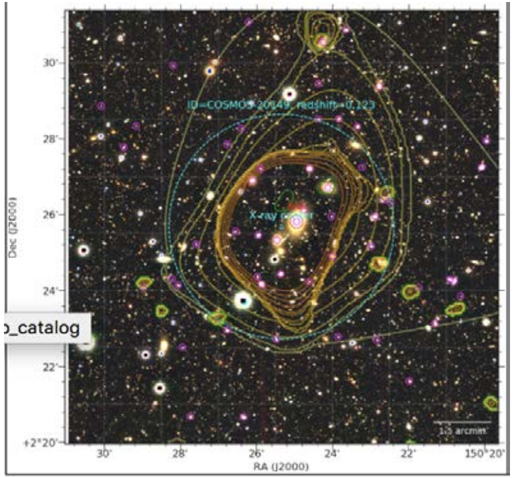
AI art and music

Deep Learning -- Success stories in science



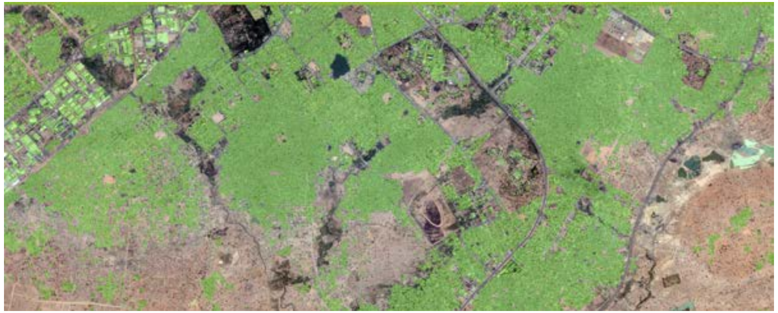
The top row shows benign tissue that has been segmented so it is easier to analyze. The bottom row shows abnormal tissue. (Courtesy of the Chinese University of Hong Kong.)

Cancer detection

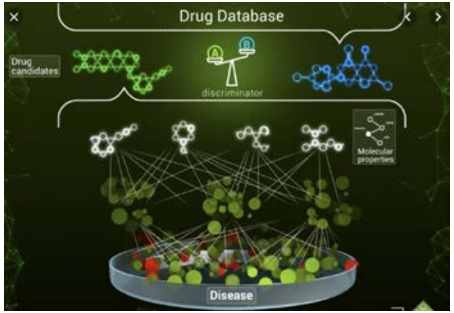


Mapping the universe

Predict protein structure

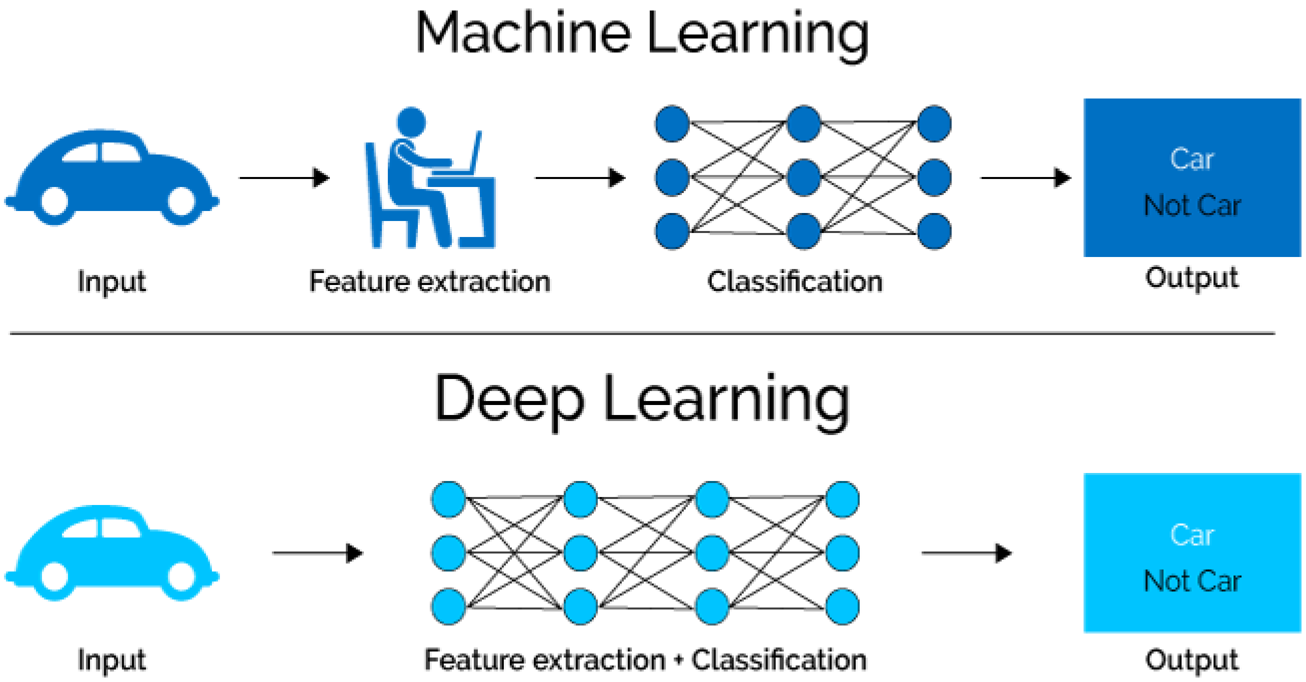


Land cover segmentation



Drug discovery

Machine Learning vs. Deep Learning



BERKELEY LAB

Fig credit: https://medium.com/intro-to-artificial-intelligence/deep-learning-series-1-intro-to-deep-learning-abb1780ee20

Outline

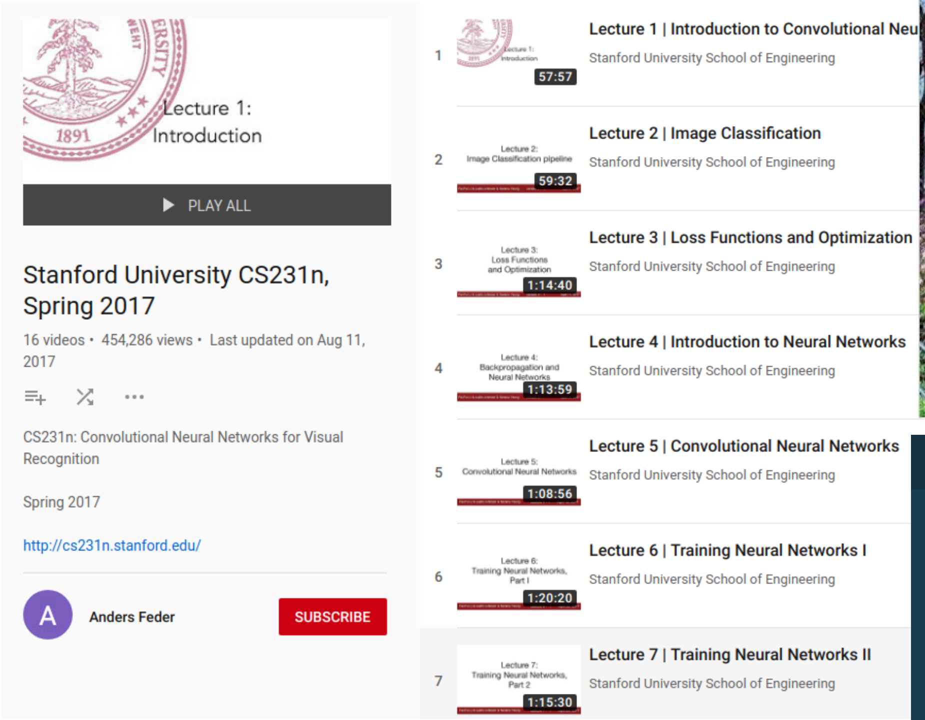
- Neural networks basics
- Neural networks optimization/training algorithms
- Monitoring neural networks training
- Convolutional neural networks basics
- Data normalization
- Learning rate decay, Batch-size schedule
- How to improve the generalization of your model? Regularization
- The importance and challenges of depth
- Transfer learning
- Some practical tips

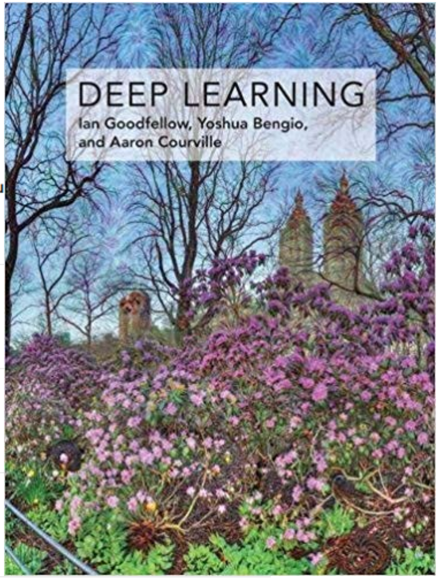
Outline

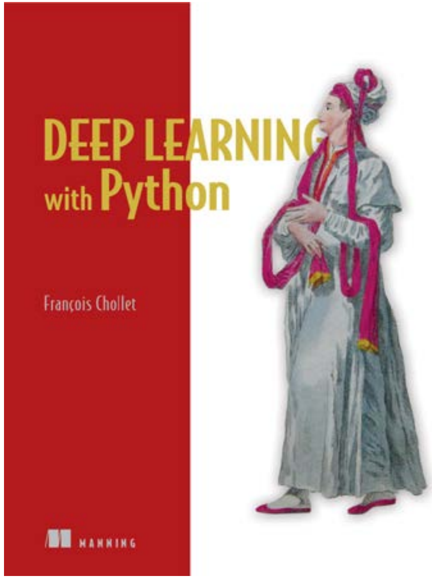
- **Neural networks basics**
- **Neural networks optimization/training algorithms**
- **Monitoring neural networks training**
- **Convolutional neural networks basics**
- Data normalization
- Learning rate decay, Batch-size schedule
- How to improve the generalization of your model? Regularization
- The importance and challenges of depth
- Transfer learning
- **Some practical tips**

Resources and acknowledgments

And hundreds of other great quality educational material and papers







Neural Networks history goes back to the 50s

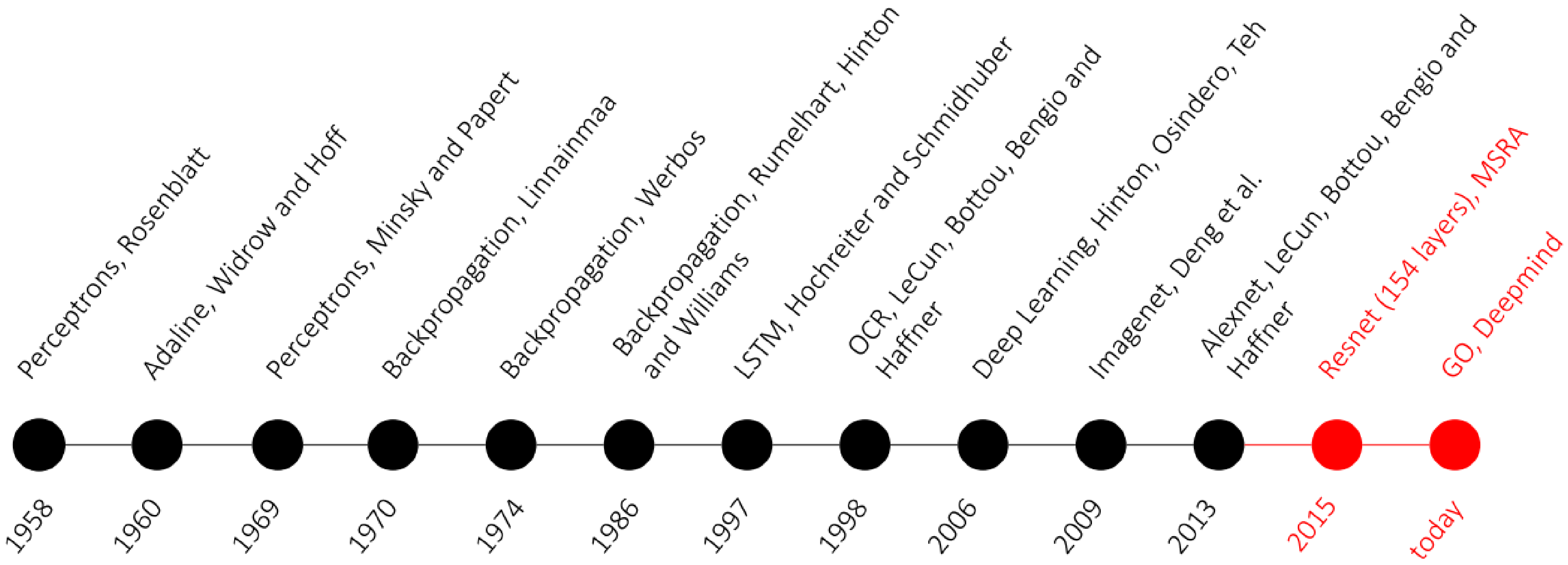
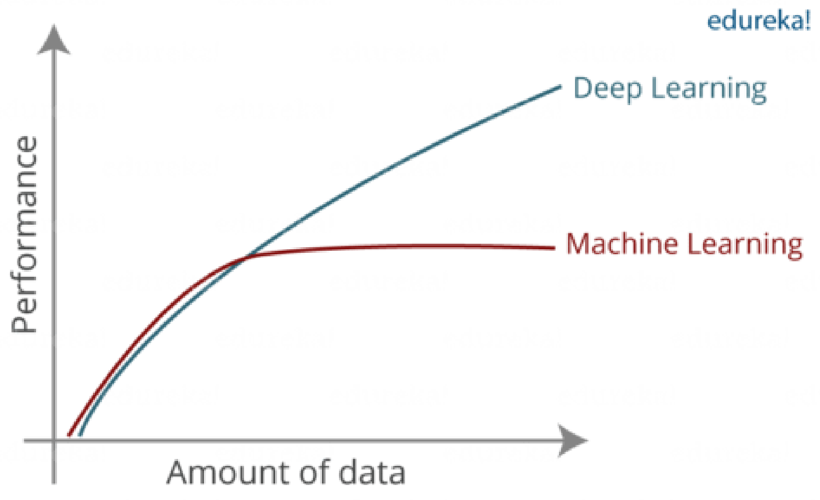


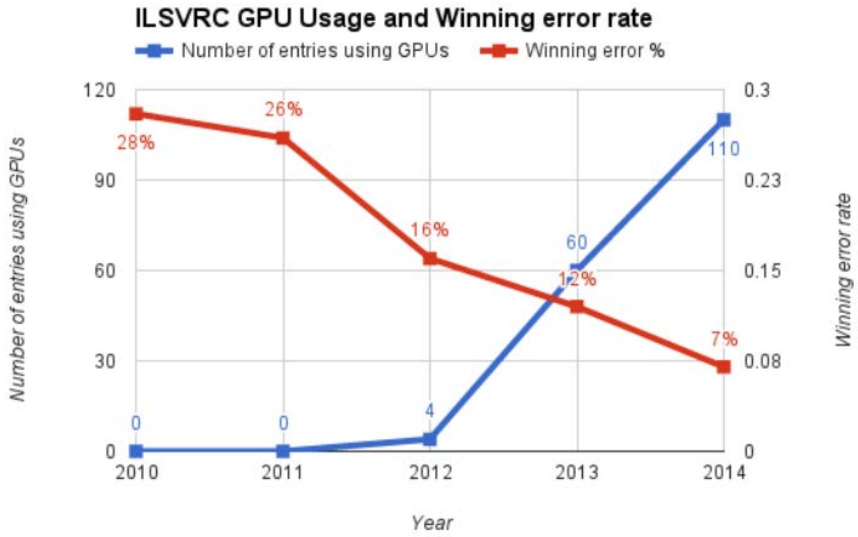
Fig. credit to Efstratios Gavves, Intro. to DL

Why do Neural Networks finally work now?

1) Data: large curated datasets



2) GPUs: linear algebra accelerators



3) Algorithmic advances: optimizers, regularization, normalization … etc.

What are Deep Neural Networks?

Long story short:

“A family of **parametric**, **non-linear** and **hierarchical** **representation learning functions**, which are massively optimized with stochastic gradient descent to **encode domain knowledge**, i.e. domain invariances, stationarity.” -- Efstratios Gavves

Neural Networks basics

Deep Forward Neural Networks (DNNs)

The objective of NNs is to approximate a function:

$$y = f^*(x)$$

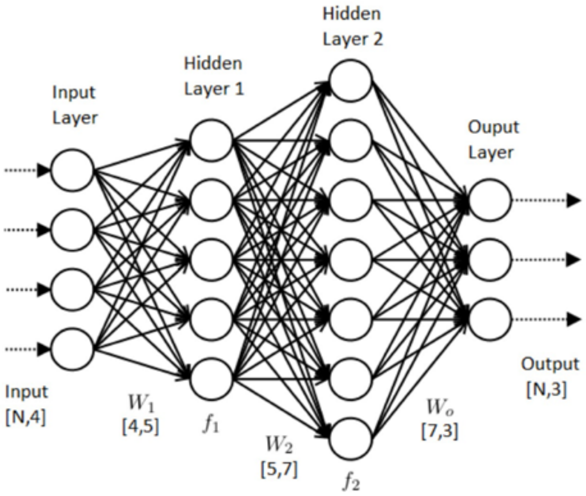
The NN learns an approximate function $y = f(x; W)$ with parameters W. This approximator is hierarchically composed of simpler functions

$$y = f^n(f^{n-1}(\cdots f^2(f^1(x))\cdots))$$

Deep Forward Neural Networks (DNNs)

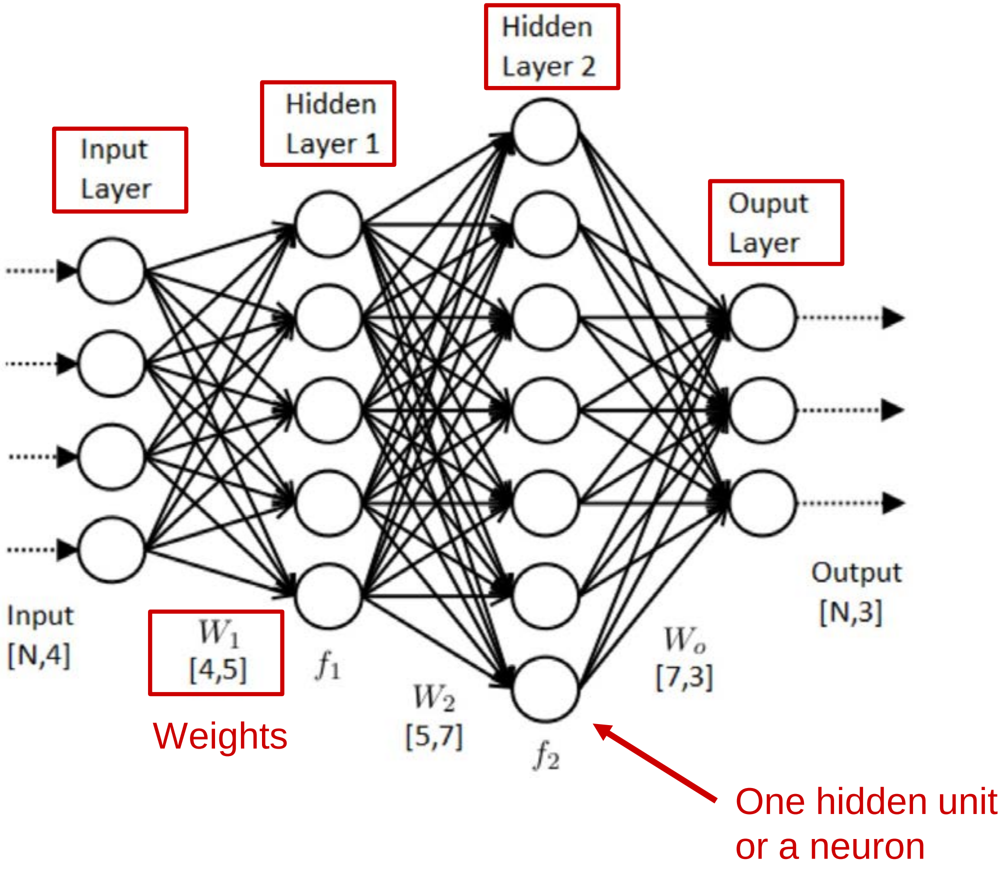
A common choice for the atomic functions is an affine transformation followed by a non-linearity (an activation function $\varphi(x)$):

$$
\begin{aligned}
h_1 &= \varphi(W_1 x + b_1) \\
h_2 &= \varphi(W_2 h_1 + b_2) \\
&\vdots \\
y &= f(h_n)
\end{aligned}
$$

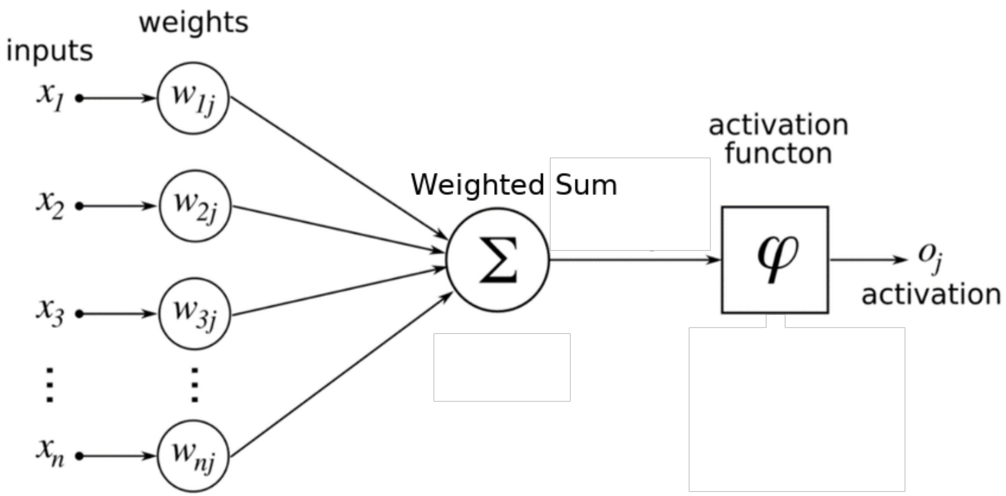


An optimization procedure is used to find network parameters, **weights Ws** and **biases bs,** that best approximate the relationship in the data, or “learn” the task.

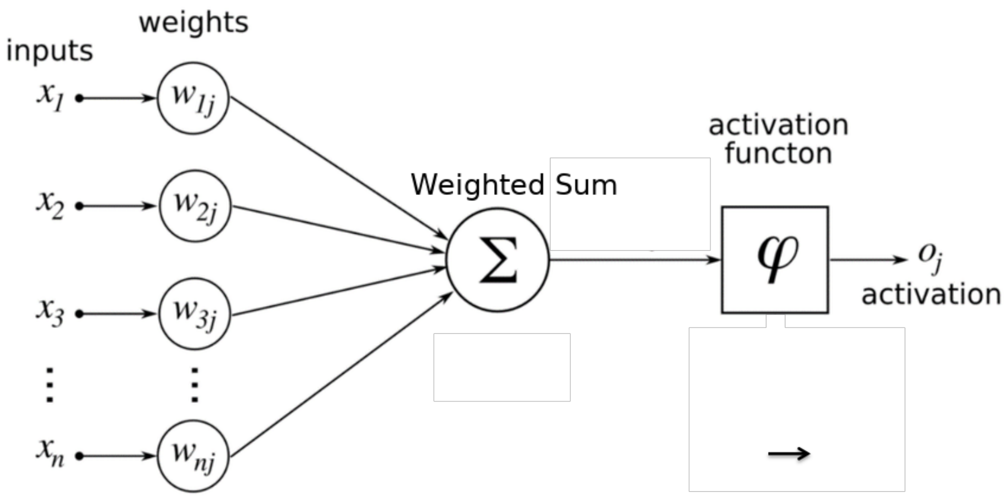
Some terminology (Fully Connected, or Dense networks)

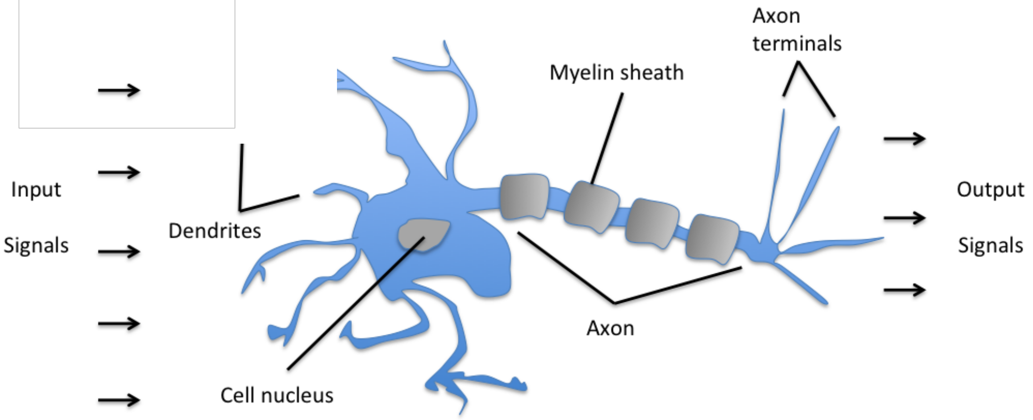


Activation functions



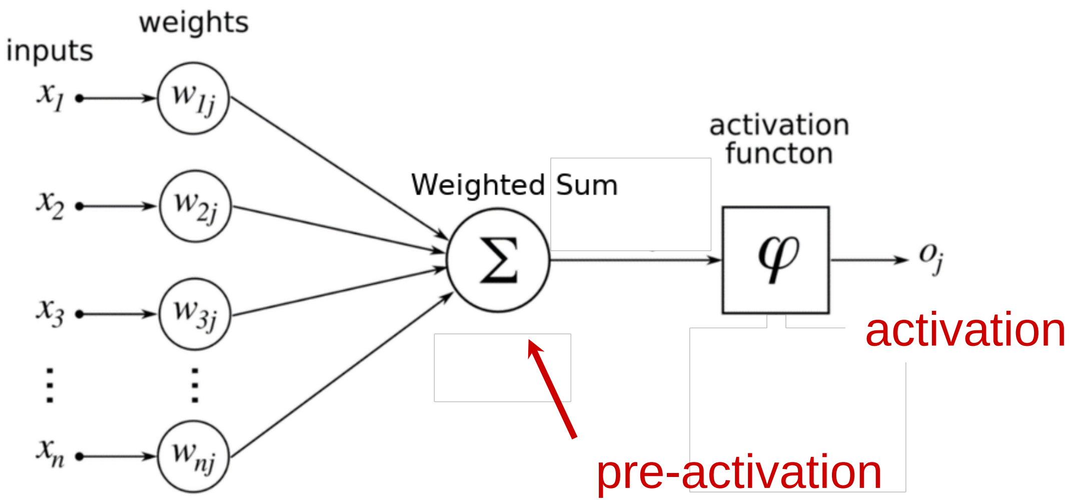
Activation functions





Schematic of a biological neuron.

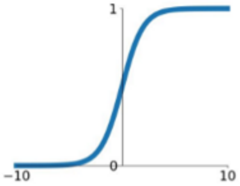
More terminology



Activation functions

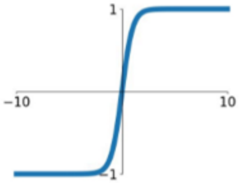
Sigmoid

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



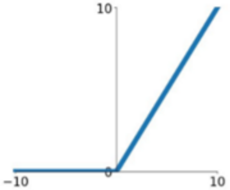
tanh

$\tanh(x)$



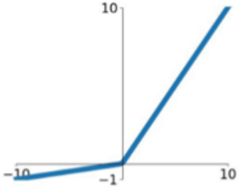
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

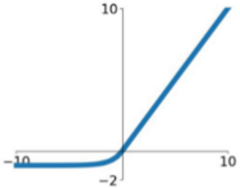


Maxout

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

ELU

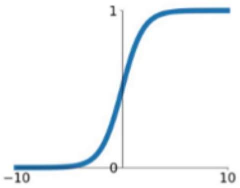
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation functions

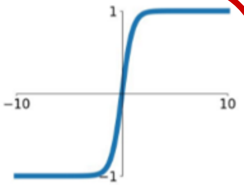
Sigmoid

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



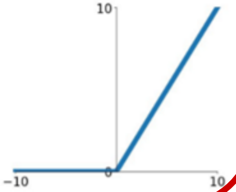
tanh

$\tanh(x)$



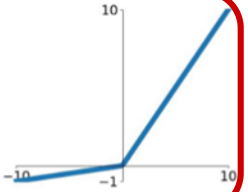
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

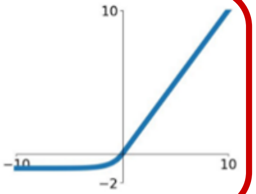


Maxout

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

ELU

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

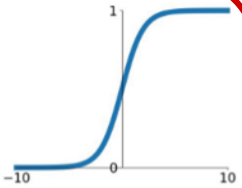


Most commonly used in modern networks as hidden layer activations

Activation functions

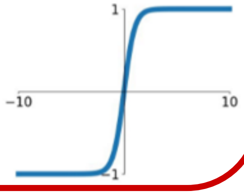
Sigmoid

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



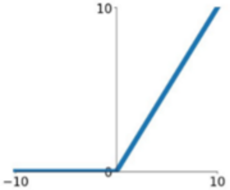
tanh

$\tanh(x)$



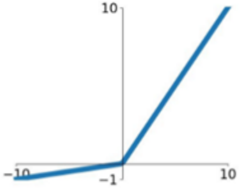
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

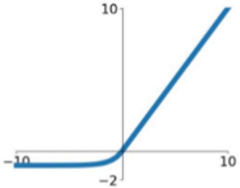


Maxout

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

ELU

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



Often used for output layers

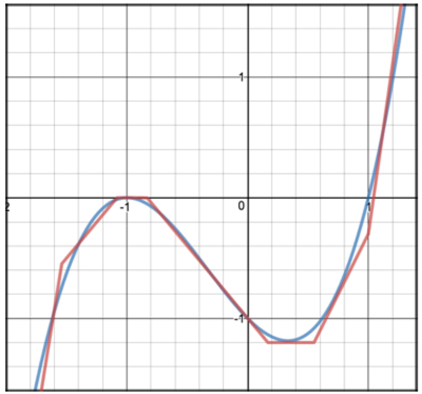
BERKELEY LAB

What kind of functions can NNs approximate?

The Universal Approximation Theorem

“a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units” -- Hornik, 1991, http://zmjones.com/static/statistical-learning/hornik-nn-1991.pdf

This, of course, does not imply that we have an optimization algorithm that can find such a function. The layer could also be too large to be practical.



$$n_1(x) = Relu(-5x - 7.7)$$
$$n_2(x) = Relu(-1.2x - 1.3)$$
$$n_3(x) = Relu(1.2x + 1)$$
$$n_4(x) = Relu(1.2x - .2)$$
$$n_5(x) = Relu(2x - 1.1)$$
$$n_6(x) = Relu(5x - 5)$$

$$Z(x) = -n_1(x) - n_2(x) - n_3(x) + n_4(x) + n_5(x) + n_6(x)$$

Fig. credit towardsdatascience.com/can-neural-networks-really-learn-any-function-65e106617fc6

Optimizing/training neural networks

Cost function & Loss

To optimize the network parameters for the task at hand we build a cost function on the training dataset:

$$J(W) = \mathbb{E}_{x,y \sim \hat{p}_{data}} L(f(x;W), y)$$

cost function: average of loss over many examples

loss function: compares model prediction to data

model prediction

Empirical Cost Minimization

The goal of machine learning is to build models that work well on unseen data, i.e. we hope to have a low cost on the data distribution p_{data} (to minimize the **true cost**)

$$J^*(W) = \mathbb{E}_{x,y \sim p_{data}} L(f(x;W), y)$$

Empirical Cost Minimization

The goal of machine learning is to build models that work well on unseen data, i.e. we hope to have a low cost on the data distribution p_{data} (to minimize the **true cost**)

$$J^*(W) = \mathbb{E}_{x,y \sim p_{data}} L(f(x;W), y)$$

However, since we don’t have access to the data distribution we resort to reducing the cost on the training dataset $\hat{p}_{data}$; i.e. minimizing the **empirical cost**, with the hope that doing so gives us a model that generalizes well to unseen data.

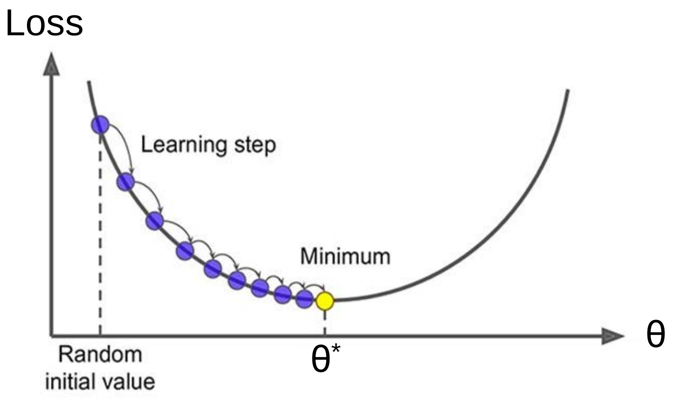
$$J(W) = \mathbb{E}_{x,y \sim \hat{p}_{data}} L(f(x;W), y)$$

Gradient Descent

Gradient descent is the dominant method to optimize network parameters θ to minimize loss function L(θ).

The update rule is **(α** is the "learning rate/step"):

$$W_{k+1} \leftarrow W_k - \alpha \nabla L(W_k)$$

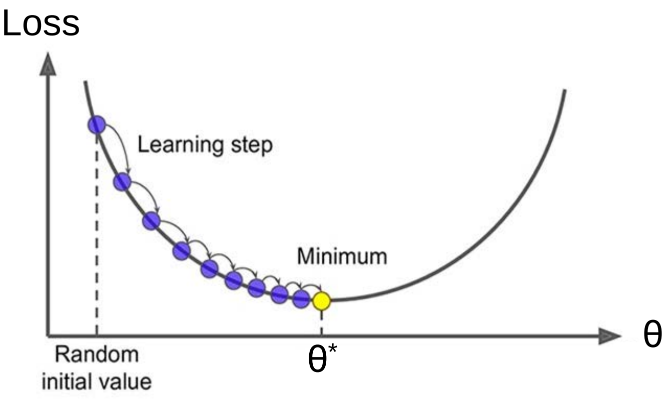


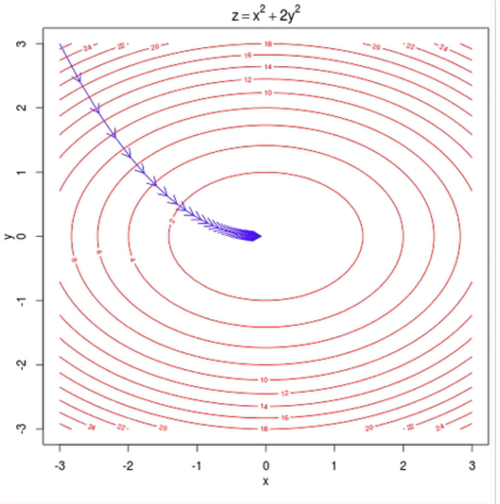
Gradient Descent

Gradient descent is the dominant method to optimize network parameters θ to minimize loss function L(θ).

The update rule is **(α is the "learning rate/step")**:

$$W_{k+1} \leftarrow W_k - \alpha \nabla L(W_k)$$





Gradient estimation: Stochastic Gradient Descent

To make a single gradient step, the gradient is taken over a “random” **minibatch** of examples **m** instead of the entire dataset

Gradient estimate

$$W_{k+1} \leftarrow W_k - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla L(x_i; W_k)$$

Gradient estimation: Stochastic Gradient Descent

To make a single gradient step, the gradient is taken over a “random” **minibatch** of examples **m** instead of the entire dataset

Gradient estimate

$$W_{k+1} \leftarrow W_k - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla L(x_i; W_k)$$

Learning rate and minibatch size are hyper-parameters

Cost function & Loss

$$J(W) = \mathbb{E}_{x,y \sim \hat{p}_{data}} L(f(x;W), y)$$

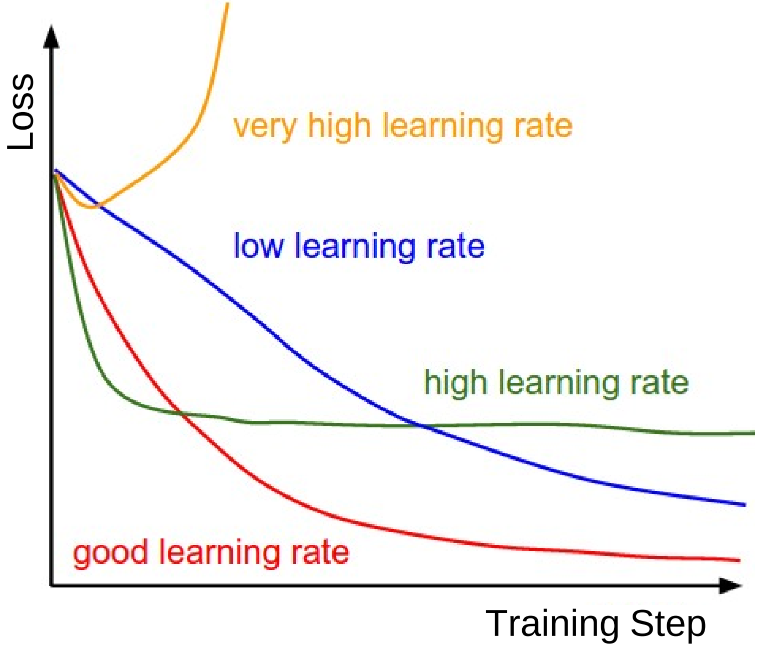
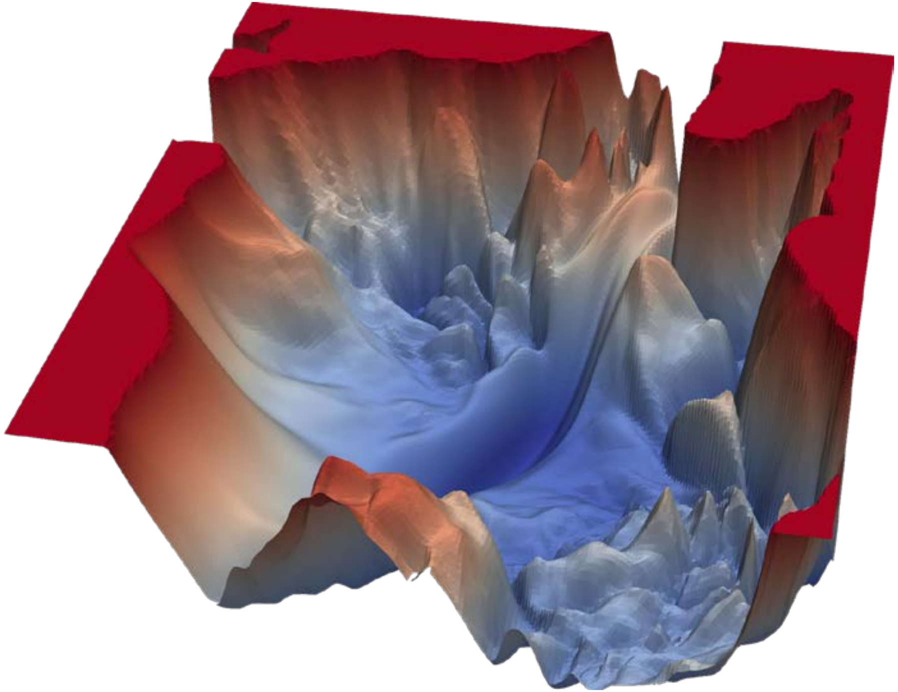


Fig. credit: http://cs231n.github.io/neural-networks-3/

Stochastic Gradient Descent variants

Gradient descent can get trapped in the abundant saddle points, ravines and local minimas of neural networks loss functions.



VGG-56 loss landscape: arXiv:1712.09913

Stochastic Gradient Descent variants

Gradient descent can get trapped in the abundant saddle points, ravines and local minimas of neural networks loss functions.

To accelerate the optimization on such functions we use a variety of methods:

- SGD + Momentum
- Nestrov
- AdaGrad
- RMSProp
- …
- Adam

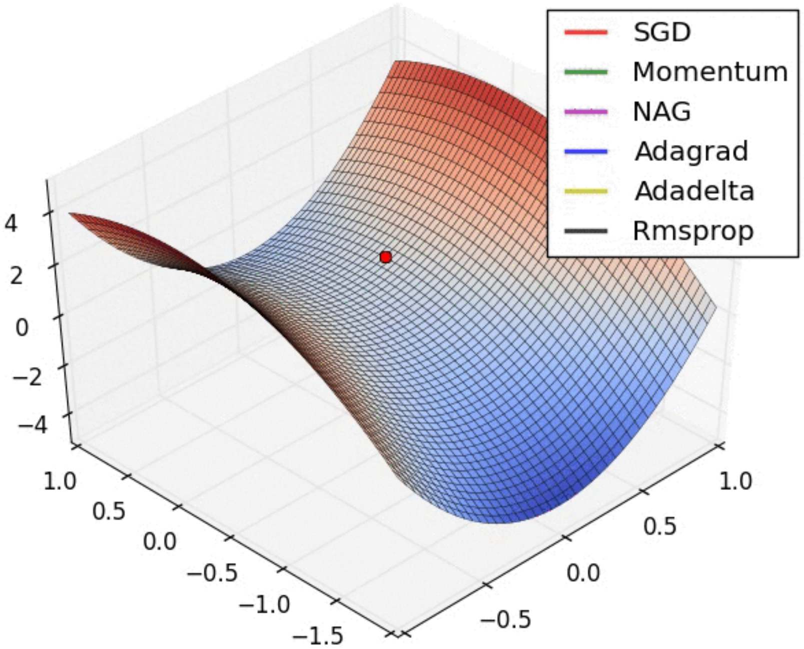
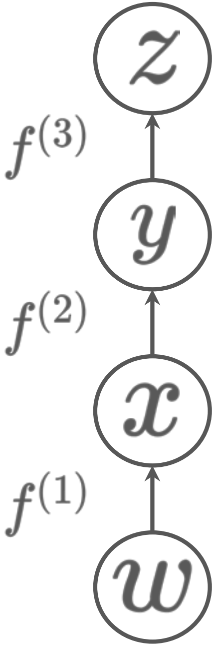


Fig. credit ruder.io/optimizing-gradient-descent/

Backpropagation

Updates to individual network parameters are propagated from the cost function through the network using the chain-rule of calculus. This is known is “backpropagation”.

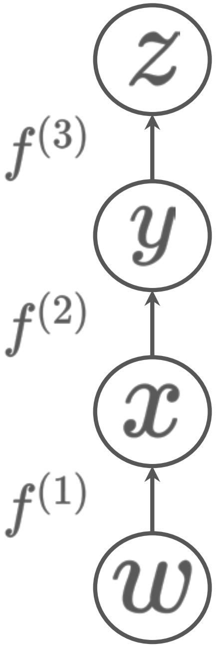


$$\frac{\partial z}{\partial w}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

$$= f^{(3)\prime}(y) f^{(2)\prime}(x) f^{(1)\prime}(w)$$

Backpropagation



$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f^{(3)\prime}(y)\, f^{(2)\prime}(x)\, f^{(1)\prime}(w)$$

Note that if any of the intermediate activations have too small derivatives or 0 (dead neurons) gradients will not flow back

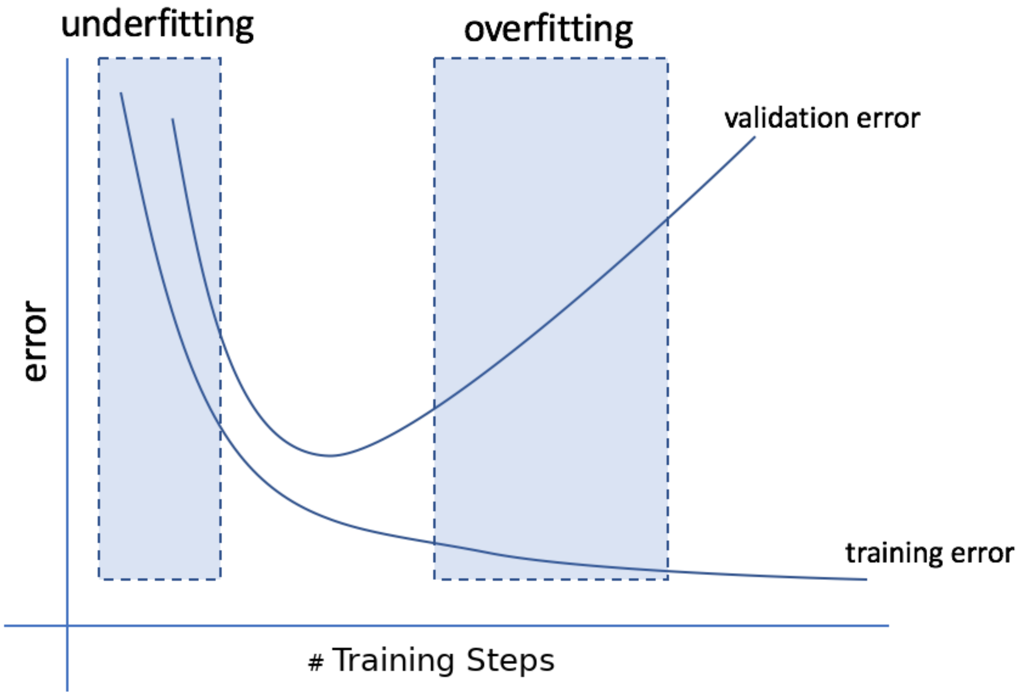
Monitoring neural networks training

Monitoring training/learning progress

Training curves are evaluated on training dataset.

Validation curves are evaluated on a development dataset.

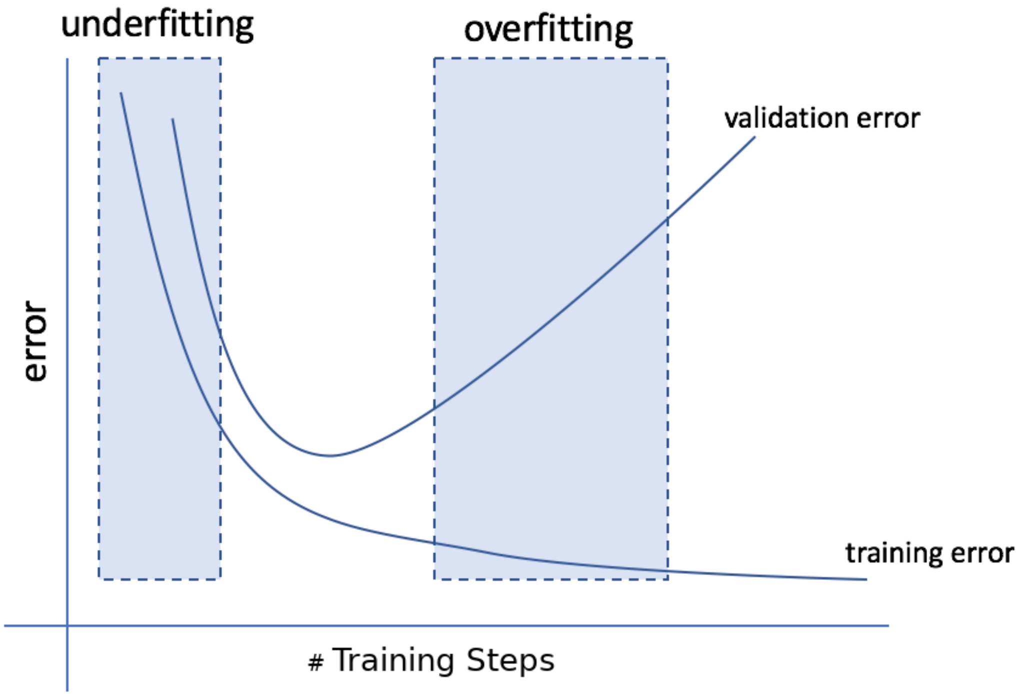
A third “test” dataset is typically held out to the very end to evaluate the performance of the final model and compare to other models.



Monitoring training/learning progress

Underfitting: training loss is high

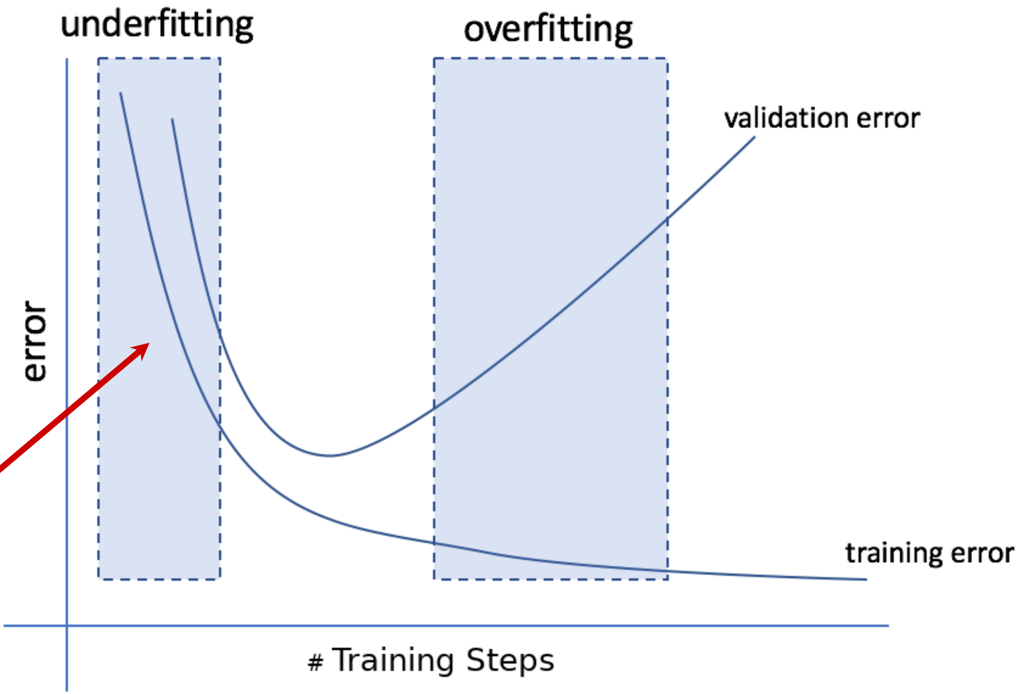
- check model architecture.
- check Learning Rate.
- train longer.
- check other hyper-parameters.



Monitoring training/learning progress

Underfitting: training loss is high

- check model architecture.
- check Learning Rate.
- train longer.
- check other hyper-parameters.

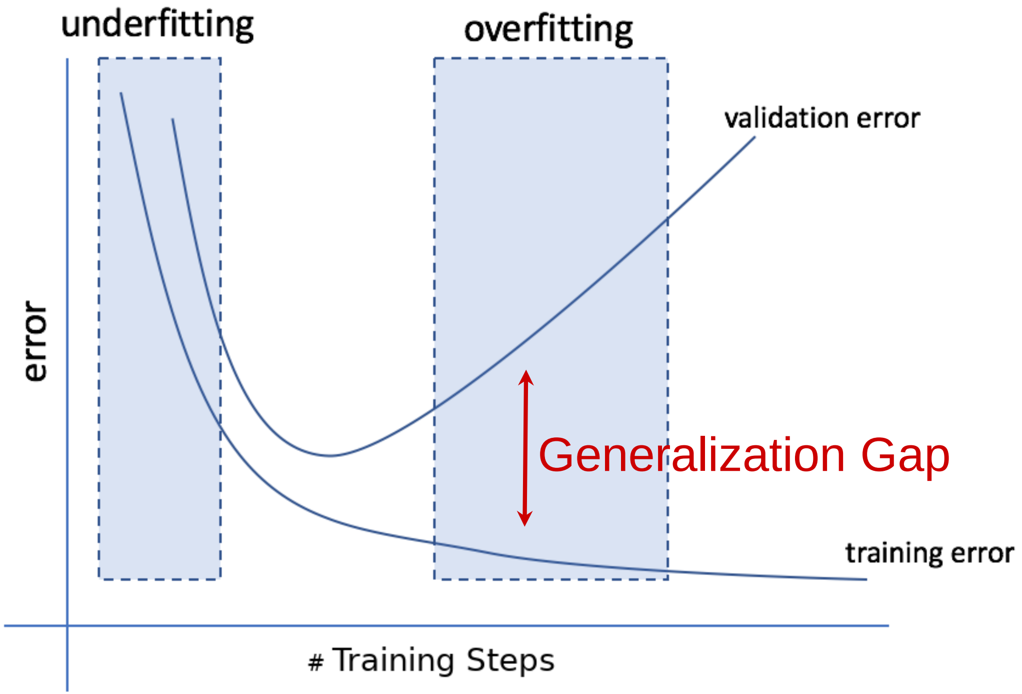


Training and validation curves (loss or accuracy) are too similar is your first clue of an underfitting problem

Monitoring training/learning progress

Overfitting: training loss is low, validation loss is high

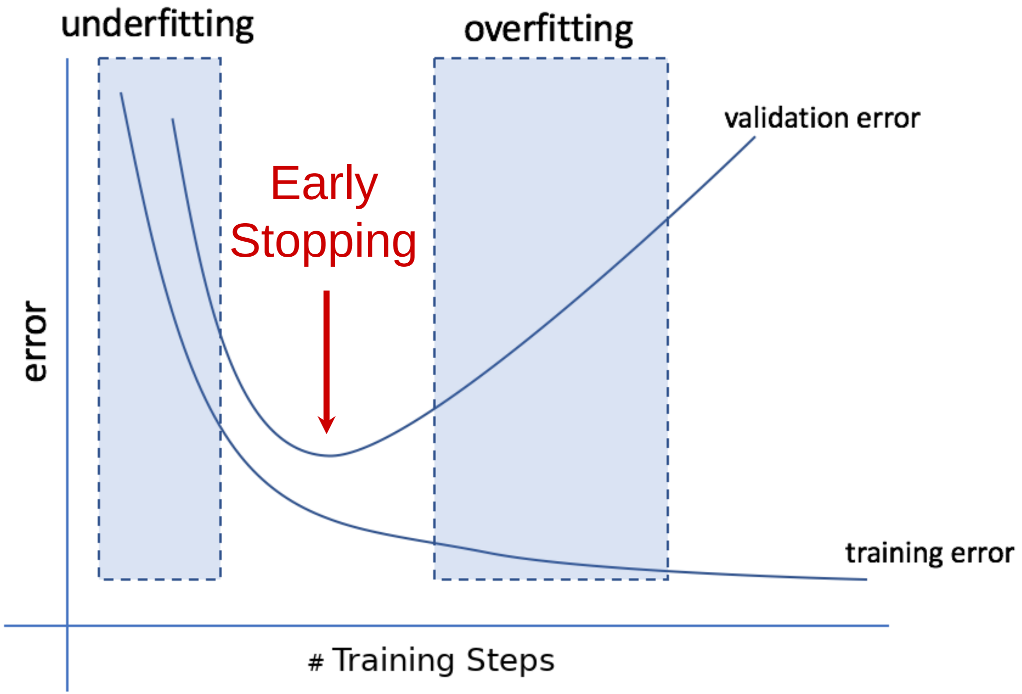
- Do you have enough data?
- Can you employ data augmentation?
- Learning-Rate tuning. Other hyper-parameters
- Regularization techniques …
- Reduce model complexity



Monitoring training/learning progress

Overfitting: training loss is low, validation loss is high

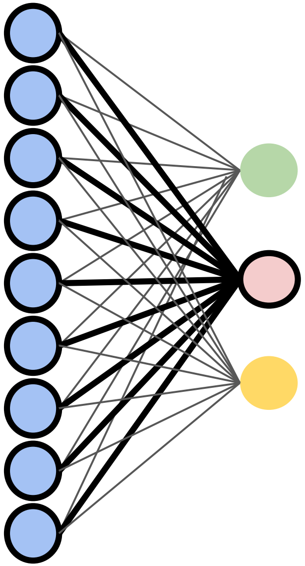
- Do you have enough data?
- Can you employ data augmentation?
- Learning-Rate tuning. Other hyper-parameters
- Regularization techniques …
- Reduce model complexity



Connectivity and Model Architecture

Connectivity

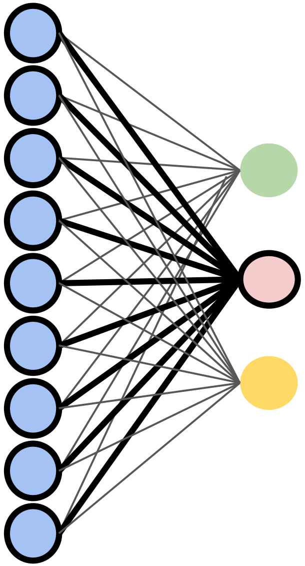
Fully Connected (Dense)



Every neuron is connected to all components of input vector.

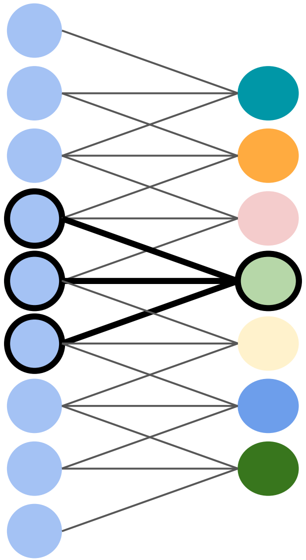
Connectivity

Fully Connected (Dense)



Every neuron is connected to all components of input vector.

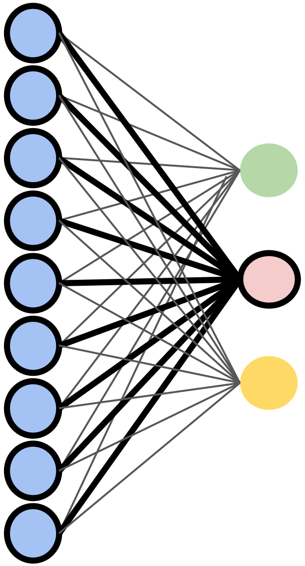
Sparse connectivity



Every neuron is only affected by a limited input “receptive field”; 3 in this example.

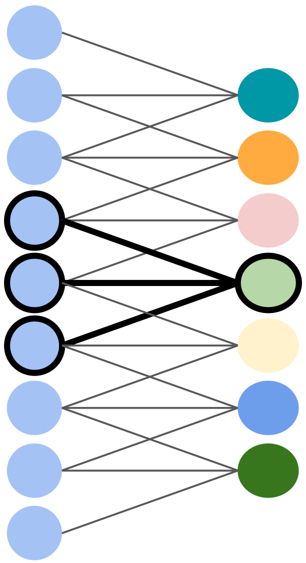
Connectivity

Fully Connected (Dense)



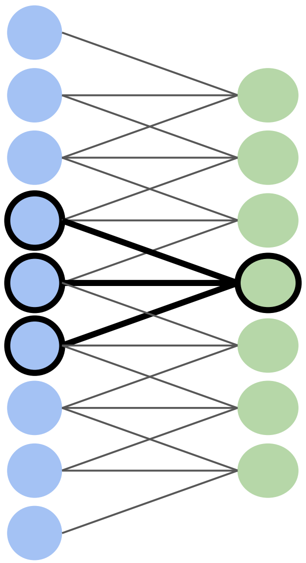
Every neuron is connected to all components of input vector.

Sparse connectivity



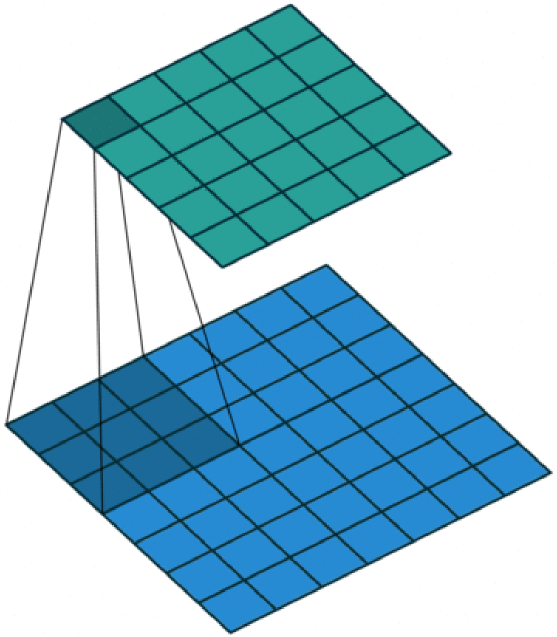
Every neuron is only affected by a limited input “receptive field”; 3 in this example.

Sparse connectivity + parameter sharing



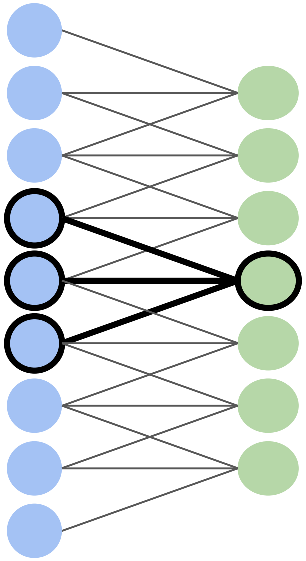
Parameters are shared (tied weights) across all neurons

Convolutional Neural Networks (CNNs)



CNNs slide the same kernel of weights across their input, thus have local sparse connectivity and tied weights

Sparse connectivity + parameter sharing



Parameters are shared (tied weights) across all neurons

BERKELEY LAB

Convolutional Neural Networks (CNNs)

CNNs implement the convolution operation over input. Sliding weights over input while computing dot product.

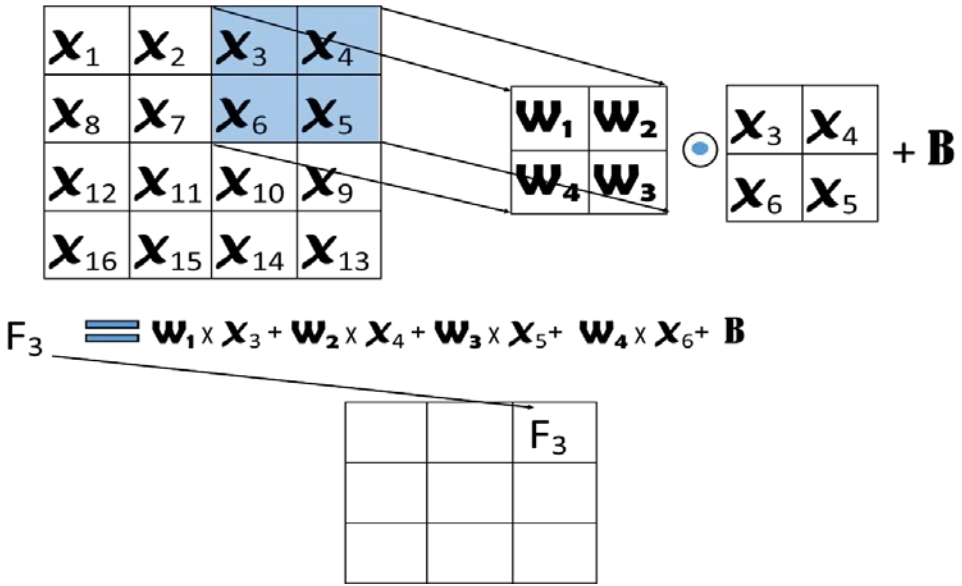
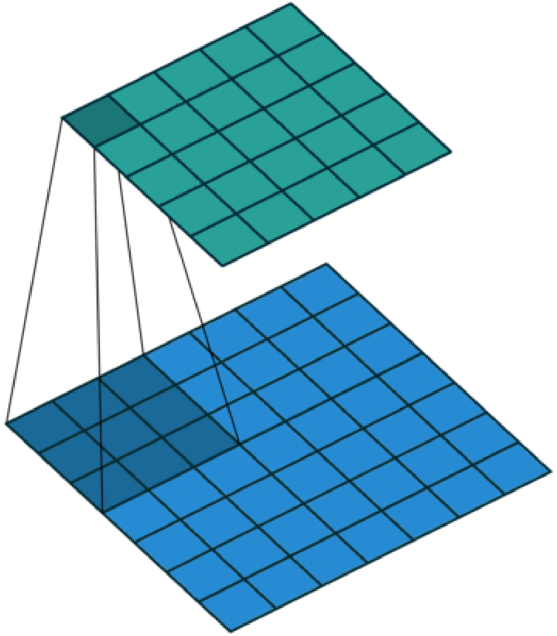


Fig. credit: goo.gl/4Qgn5U

Convolutional Neural Networks (CNNs)

CNNs are translation equivariant by construction.

CNNs achieve: sparse connectivity, parameter sharing and translation equivariance.



Sliding convolution kernel with size 3x3 over an input of 7x7.

Fig. credit: github.com/vdumoulin/conv_arithmetic

More terminology

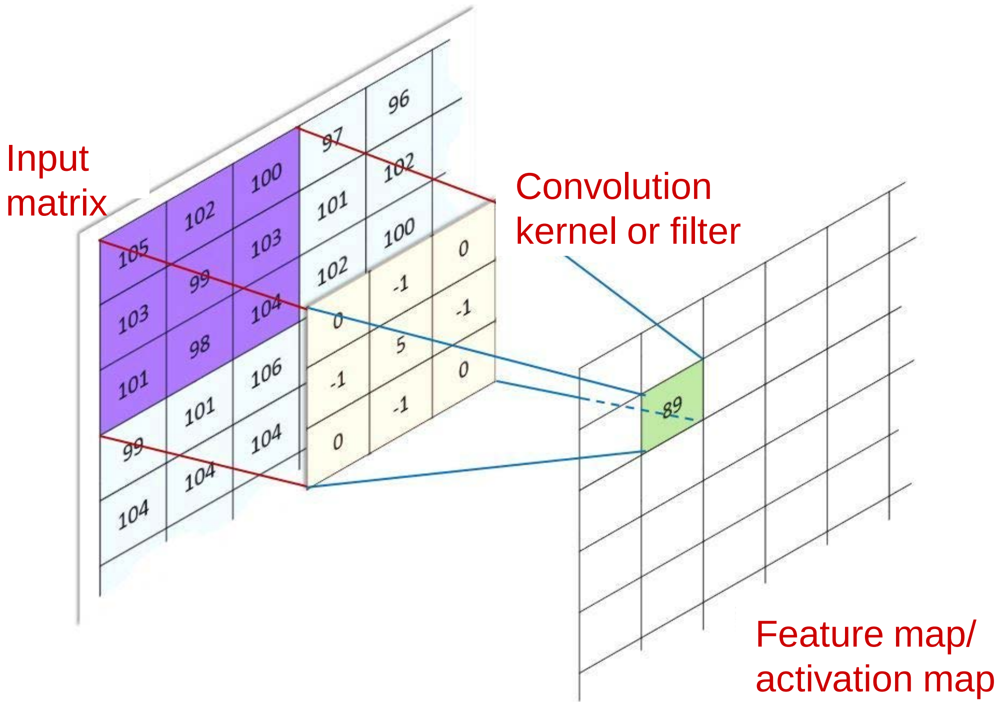


Fig. credit: brilliant.org/wiki/convolutional-neural-network/

CNNs output dimensions

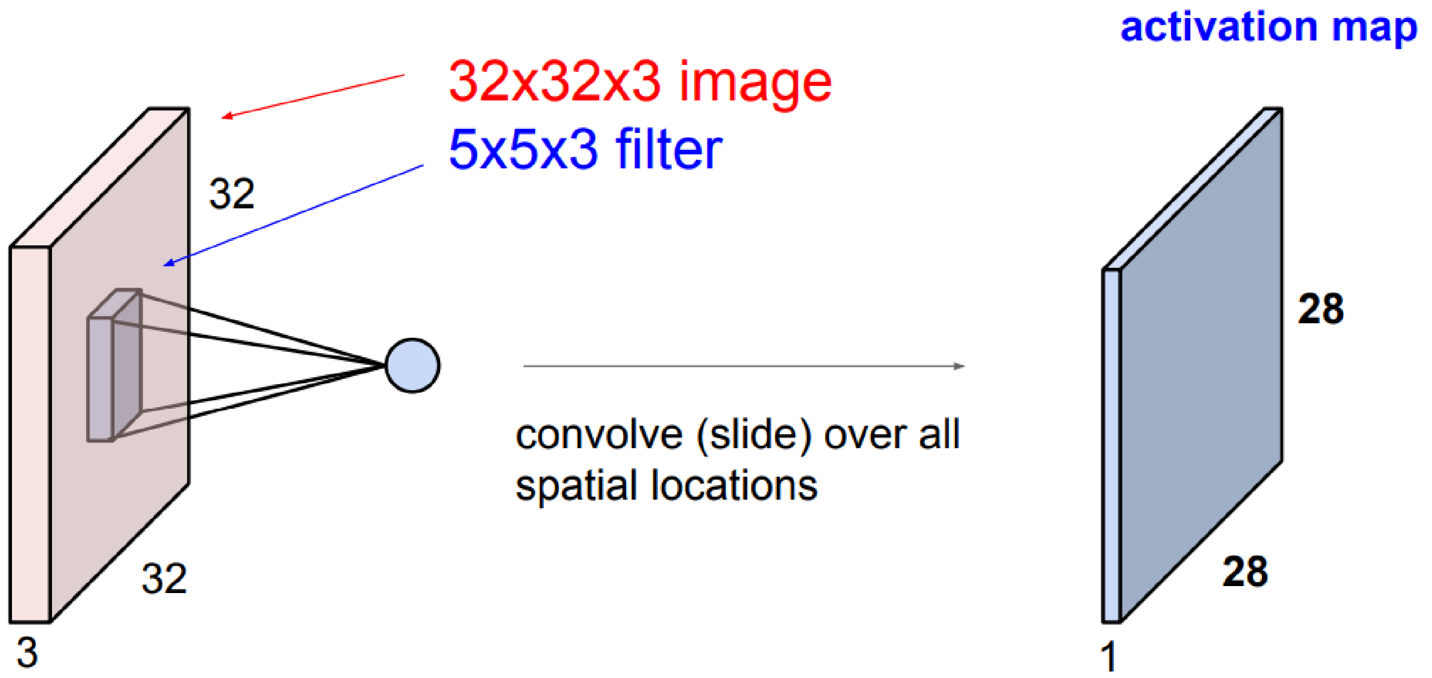
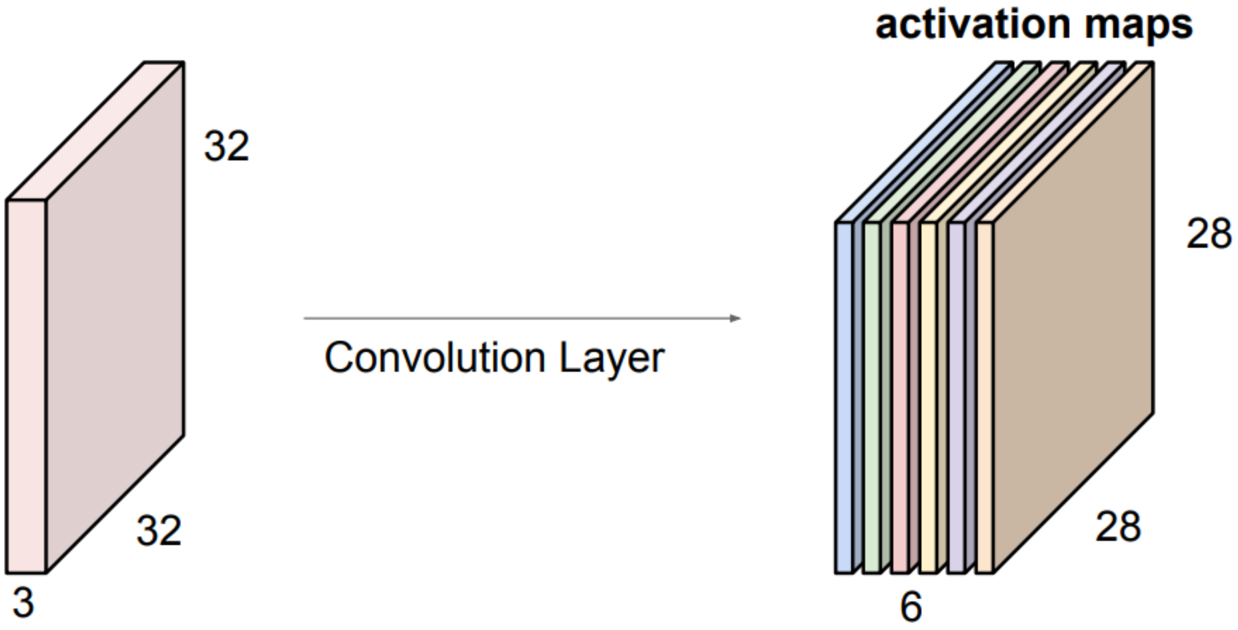


Fig. credit: [Lecture 5, CS231 Spring 2019]

CNNs output dimensions

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

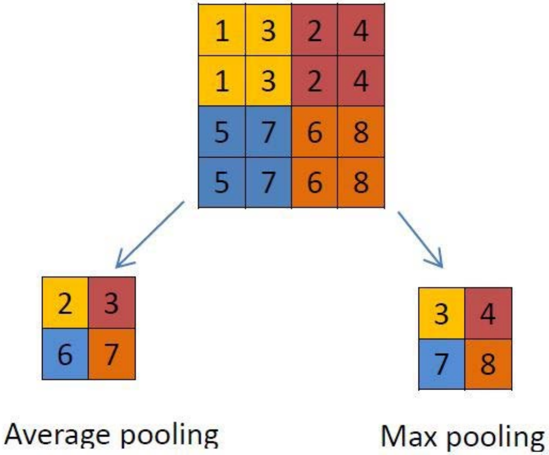


We stack these up to get a "new image" of size 28x28x6!

Fig. credit: Lecture 5, CS231 Spring 2019

Pooling

Pooling with kernel size 2



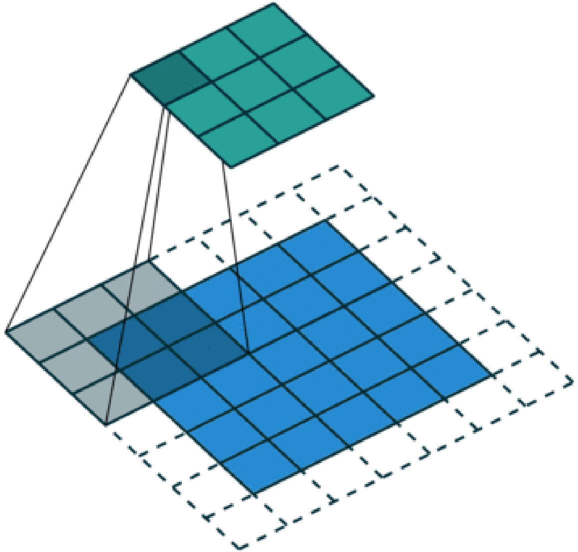
Pooling layers replace their input by a summary statistic of the nearby pixels.

Max-pool and Avg-pool are the most common pooling layers.

Pooling helps make the model invariant to small local translations of input.

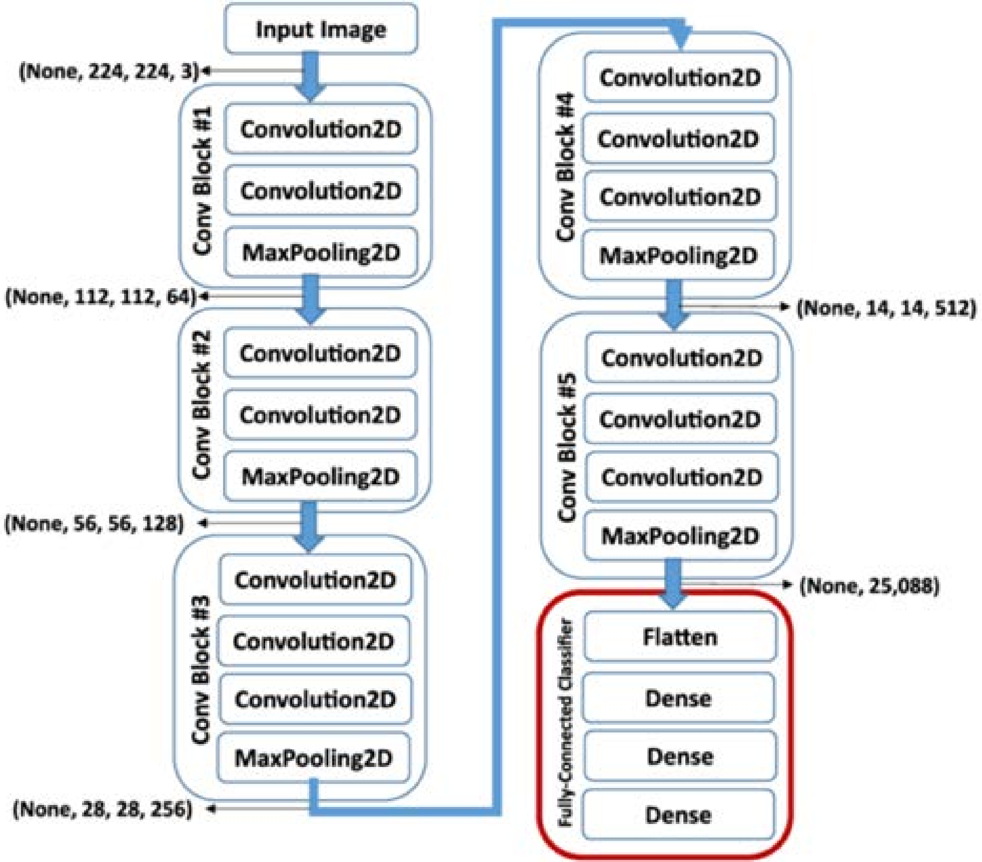
Strided convolutions

Kernel=3, stride=2 convolution



Strided convolutions are another way to reduce the spatial dimensionality of the feature maps, the intuition in using strided convolutions is to let the network learn the proper “pooling” function.

Let us put it all together: a typical CNN network architecture



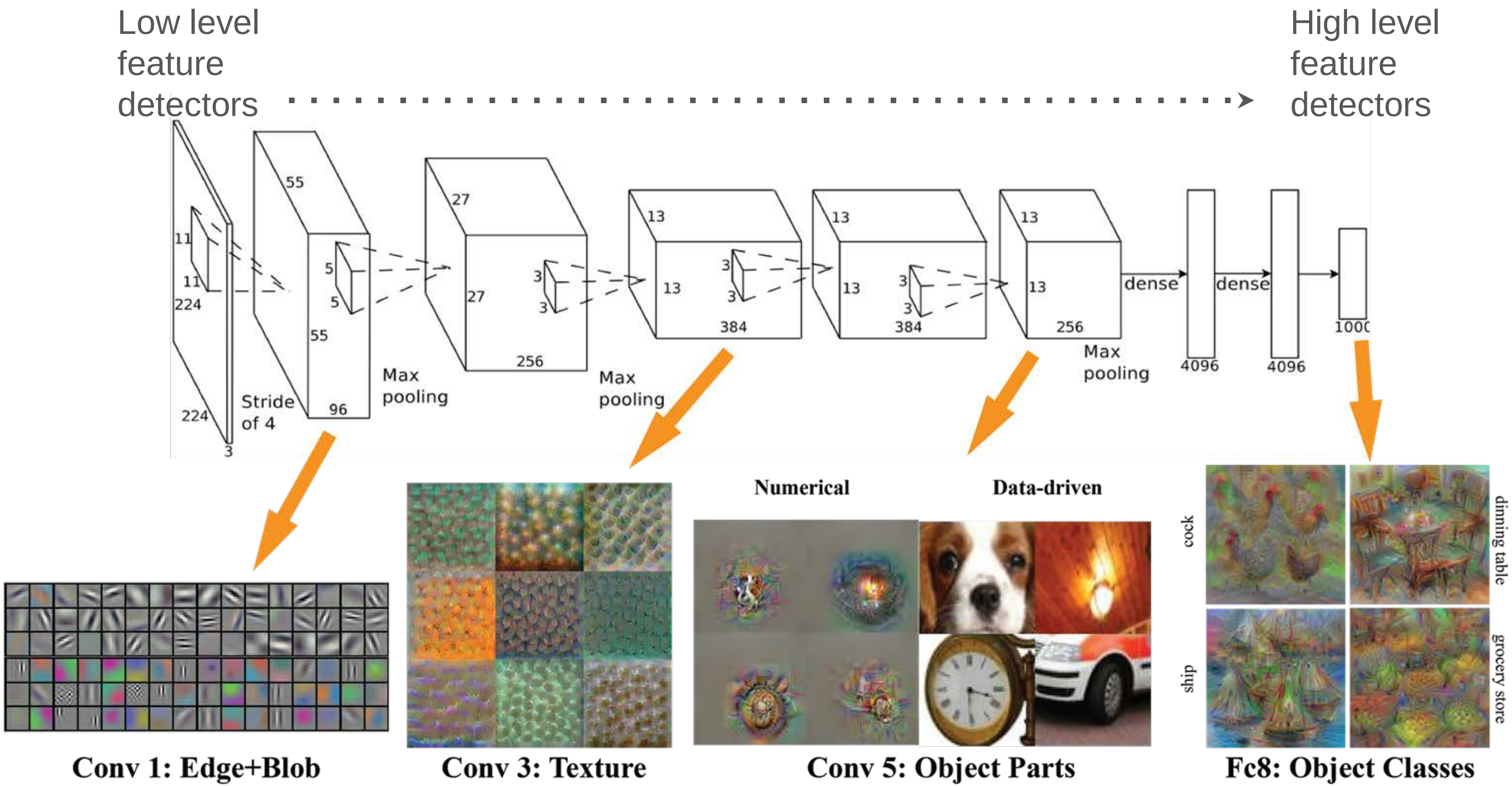
A schematic of VGG-16 Deep Convolutional Neural Network (DCNN) architecture trained on ImageNet (2014 ILSVRC winner)

And there's more!

- Recurrent networks
  - Modeling time
- Transposed Convolutions
  - For image generation (also known as *upsampling*)
- Skip connections
  - Helps to train really massive networks
- Geometric Deep Learning
  - Spherical convolutions, modeling groups, flows, etc
- And more!

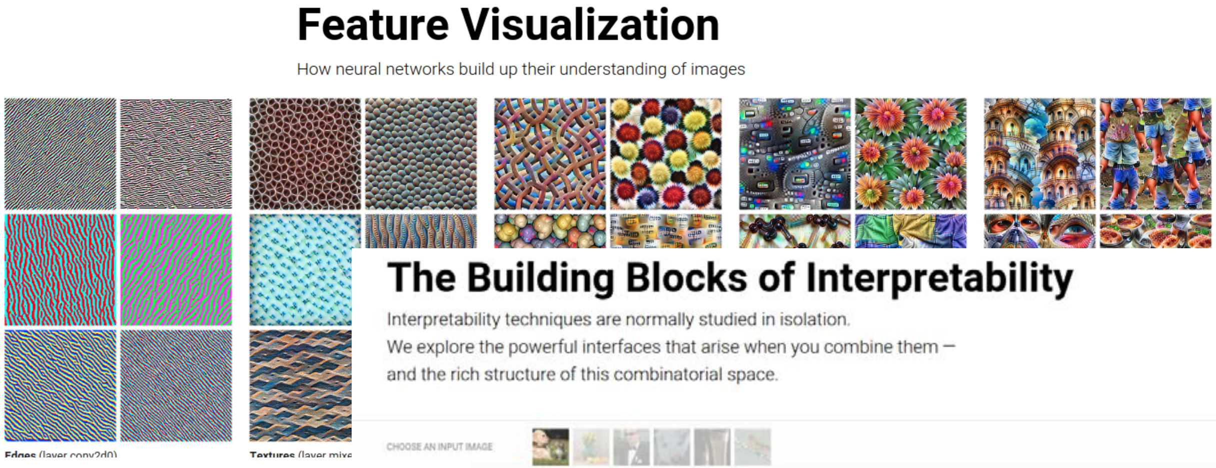
Demystifying the black box

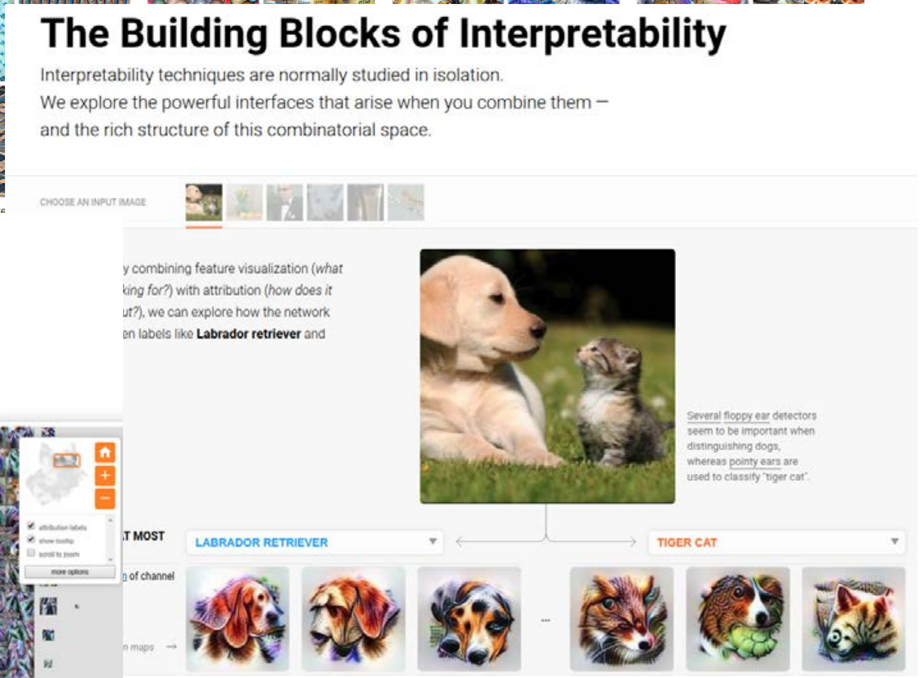
What do CNNs “learn”? Feature visualization

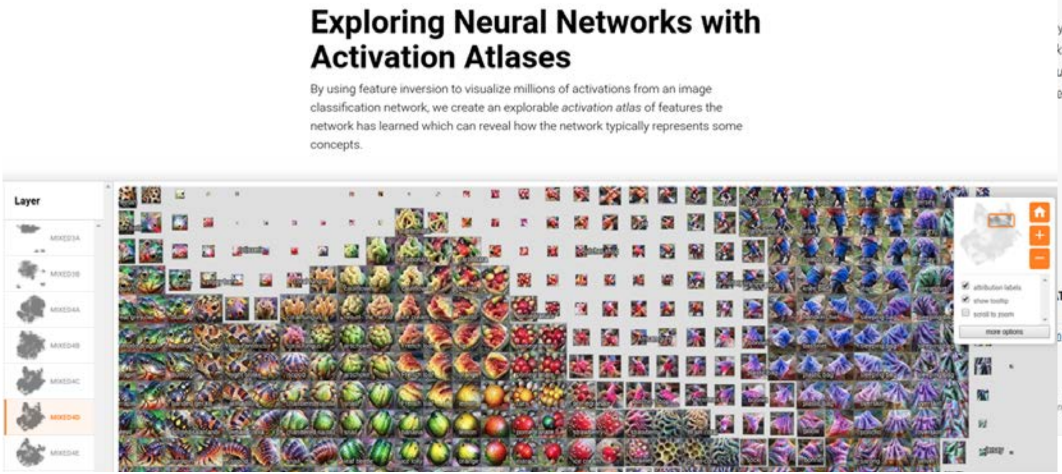


mNeuron: A Matlab Plugin to Visualize Neurons from Deep Models vision03.csail.mit.edu/cnn_art/index.html

Checkout these articles by Chris Olah et al on dstill.pub







What are Deep Neural Networks?

Long story short:

"A family of **parametric**, **non-linear** and **hierarchical** **representation learning functions**, which are massively optimized with stochastic gradient descent to **encode domain knowledge**, i.e. domain invariances, stationarity." -- Efstratios Gavves

A couple of practical tips

Check loss at the beginning of training

When you start from randomly initialized weights you can expect your network to give random chance outputs, a helpful debugging step is to make sure the value of the loss function at beginning of the training makes sense.

For example, if you are using a negative log-likelihood for a 10-classes classification problem you expect you first loss to be $\sim -\log(1/C) = -\log(1/10) \approx 2.3$

```
model.fit(train_images[0:32], train_labels[0:32], batch_size=32, epochs=1)
```

```
32/32 [==============================] - 0s 2ms/sample - loss: 2.3919 - acc: 0.0625
<tensorflow.python.keras.callbacks.History at 0x7f4b7967f6d8>
```

Remember to turn off any regularization for this check.

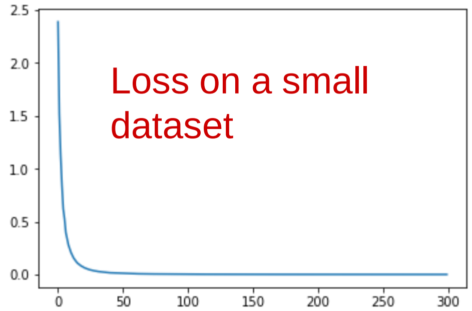
Make sure your network can overfit a tiny dataset first

Neural networks are over-parameterized functions, your model should have the representational capacity to overfit a tiny dataset. This is the first thing you should check. **If your model can't achieve a ~ 100% accuracy on a small dataset there is no point of trying to "learn" on the full dataset.** Stop and debug your code!

Tiny dataset

```
h = model.fit(train_images[0:32], train_labels[0:32], batch_size=4, epochs=1000, verbose=1)

Epoch 1/1000
32/32 [==============================] - 0s 3ms/sample - loss: 2.3887 - acc: 0.1562
Epoch 2/1000
32/32 [==============================] - 0s 501us/sample - loss: 1.5197 - acc: 0.5312
Epoch 3/1000
32/32 [==============================] - 0s 717us/sample - loss: 1.1332 - acc: 0.6875
Epoch 4/1000
32/32 [==============================] - 0s 643us/sample - loss: 0.8480 - acc: 0.8438
Epoch 5/1000
32/32 [==============================] - 0s 766us/sample - loss: 0.6225 - acc: 0.9062
Epoch 6/1000
32/32 [==============================] - 0s 608us/sample - loss: 0.5281 - acc: 0.9062
Epoch 7/1000
32/32 [==============================] - 0s 522us/sample - loss: 0.3970 - acc: 0.9688
Epoch 8/1000
32/32 [==============================] - 0s 502us/sample - loss: 0.3396 - acc: 1.0000
Epoch 9/1000
```



Loss on a small dataset

100% accuracy

Andrej Karpathy blog

About   Hacker's guide to Neural Networks

A Recipe for Training Neural Networks

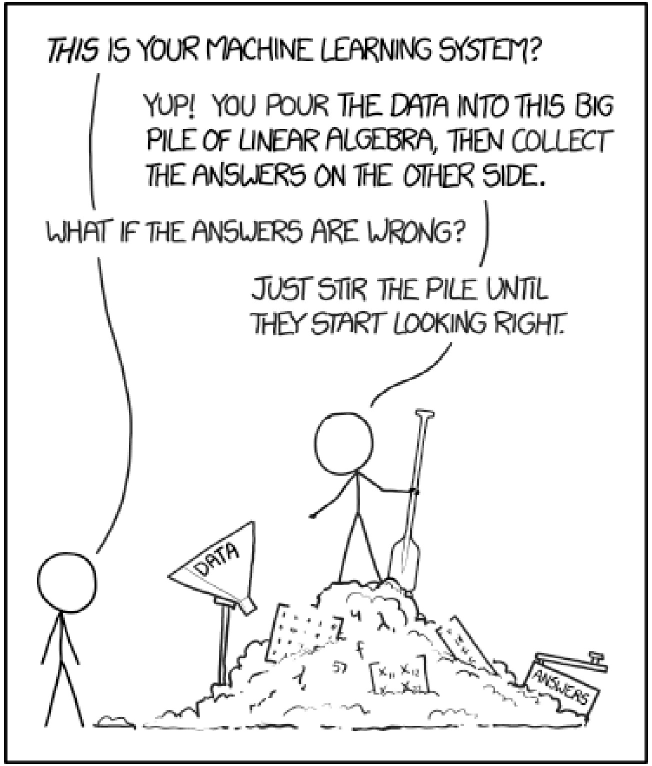
Apr 25, 2019

Some few weeks ago I posted a tweet on "the most common neural net mistakes", listing a few common gotchas related to training neural nets. The tweet got quite a bit more engagement than I anticipated (including a webinar :)). Clearly, a lot of people have personally encountered the large gap between "here is how a convolutional layer works" and "our convnet achieves state of the art results".

So I thought it could be fun to brush off my dusty blog to expand my tweet to the long form that this topic deserves. However, instead of going into an enumeration of more common errors or fleshing them out, I wanted to dig a bit deeper and talk about how one can avoid making these errors altogether (or fix them very fast). The trick to doing so is to follow a certain process, which as far as I can tell is not very often documented. Let's start with two important observations that motivate it.

http://karpathy.github.io/2019/04/25/recipe/

xkcd.com/1838/



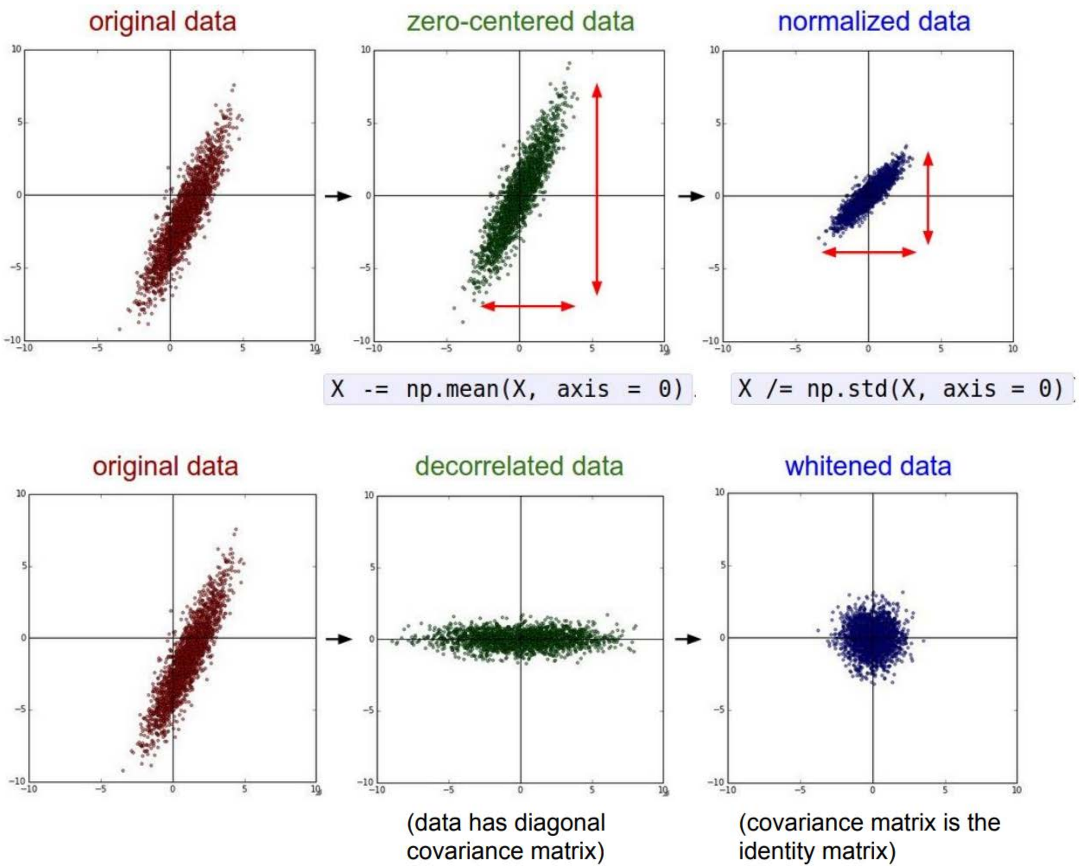
BUT this stirring is damn hard!

Thank You

Data preprocessing

You don't want your model to be too sensitive to the relative scales, if that is irrelevant.

"It only makes sense to apply this preprocessing if you have a reason to believe that different input features have different scales (or units), but they should be of approximately equal importance to the learning algorithm" -- CS231n notes

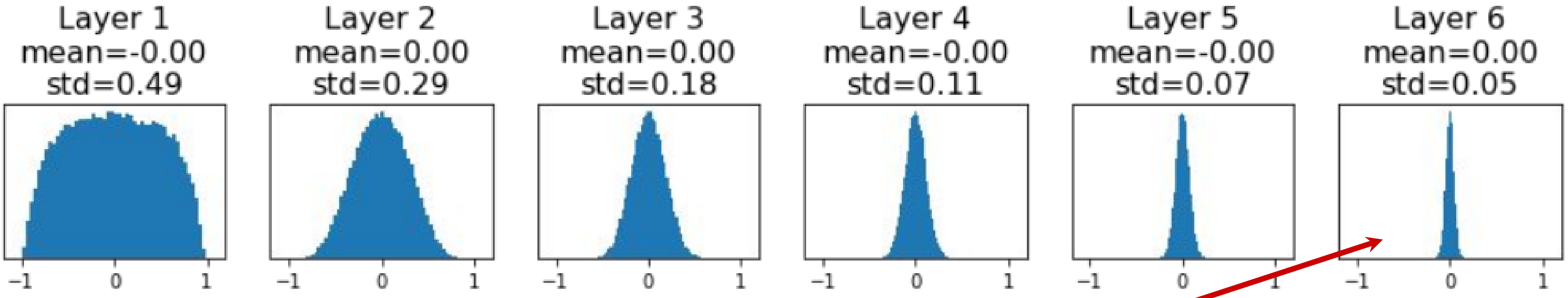


Weights initialization

Neural networks weights have to be randomly initialized to break the symmetry

Normal distribution initialization with a constant σ works okay for small networks but kills gradients for deeper networks

For example, take initialization $W \sim 0.01 \times \mathcal{N}(0, 1)$



Tiny activations → tiny gradients

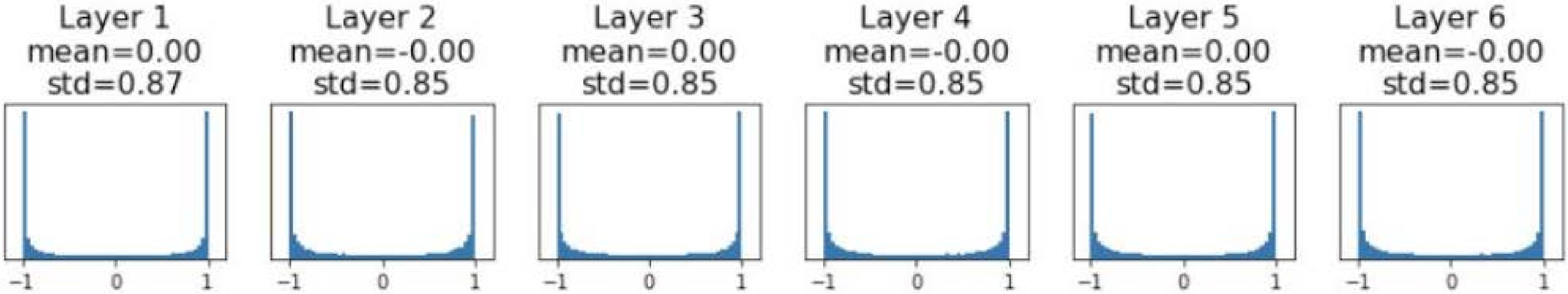
Example credit: CS231n, Spring 2019, Lecture 7

Weights initialization

Neural networks weights have to be randomly initialized to break the symmetry

Normal distribution initialization with a constant σ works okay for small networks but kills gradients for deeper networks

For example, take initialization $$W \sim 0.05 \times \mathcal{N}(0,1)$$



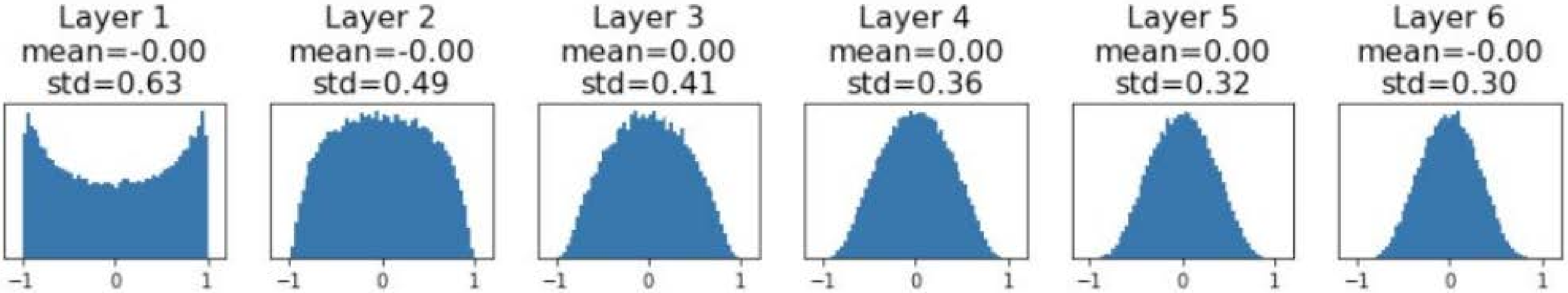
Activation saturate → zero gradients

Example credit: [CS231n, Spring 2019, Lecture 7]

Weights initialization

Xavier initialization gets around this issue:

$$W \sim \frac{1}{\sqrt{d_{in}}} \times \mathcal{N}(0,1)$$



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Decaying learning rate

There are various **learning rate schedules** that are common in practice:

- Linear decay
- Exponential decay
- Cosine
- Inverse square-root

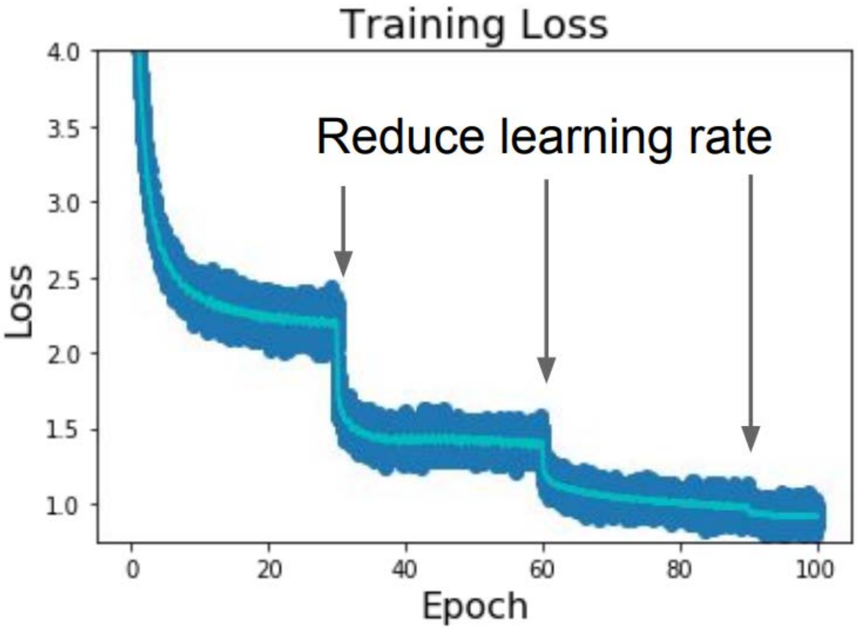
These are applied as a function of SGD step or epoch. For example, exponential decay would be:

$$\alpha = \alpha_0(1 - \frac{t}{T})$$

where:

α_0 is the initial learning rate
t is the step or epoch number
T is the total number of steps or epochs.

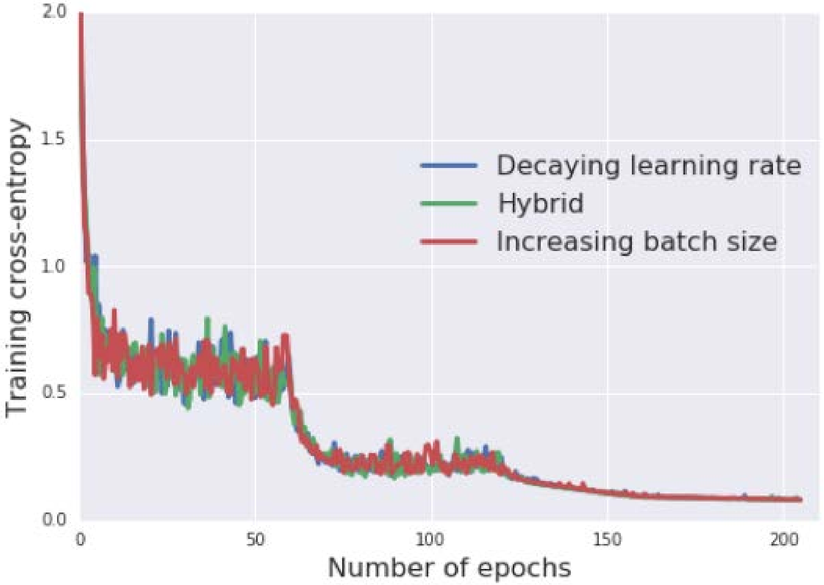
Decaying learning rate




```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                              patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

Another approach is to monitor the training curves and reduce the learning rates on plateaus, e.g. divide learning rate by 10 when validation error plateaus

An alternative could be to increase the batch-size



arXiv.org > cs > arXiv:1711.00489

Computer Science > Machine Learning

Don't Decay the Learning Rate, Increase the Batch Size

Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, Quoc V. Le

Regularization

Remember that the goal of learning, as opposed to traditional optimization, is to do well on an unseen data rather than too well on the training dataset.

We use regularization to try to prevent the model from overfitting the training dataset

$$J(W, \lambda) = \underbrace{\frac{1}{m} \sum_i L(x_i; W)}_{\text{Minibatch loss: try to fit well to the training data}} + \underbrace{\lambda \Omega(W)}_{\text{Parameter norm penalty: don't fit too well to the training data}}$$

Minibatch loss: try to fit well to the training data

Parameter norm penalty: don't fit too well to the training data

Regularization

Remember that the goal of learning, as opposed to traditional optimization, is to do well on an unseen data rather than too well on the training dataset.

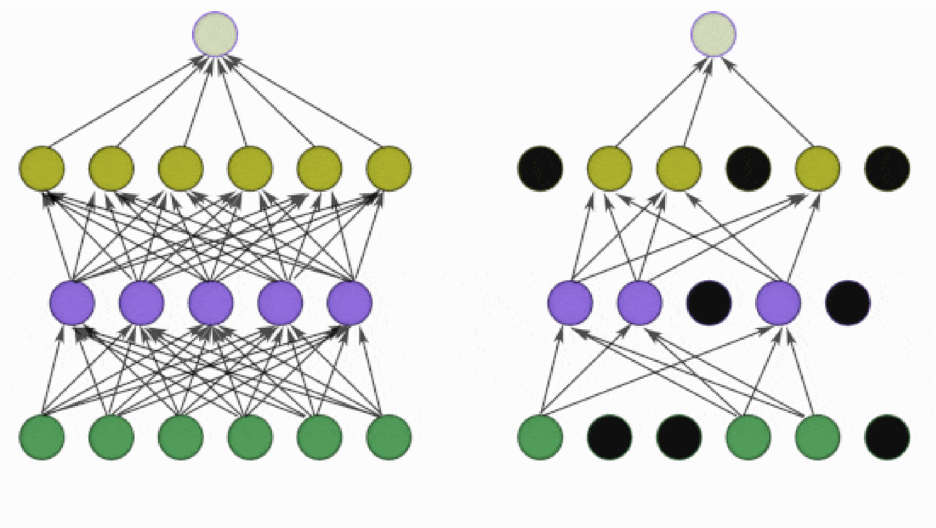
We use regularization to try to prevent the model from overfitting the training dataset

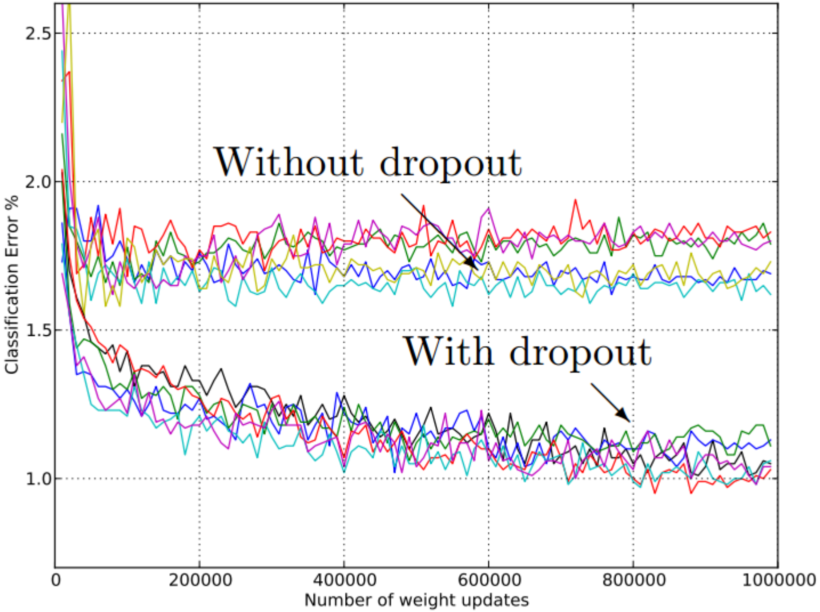
$$J(W, \lambda) = \underbrace{\frac{1}{m} \sum_i L(x_i; W)}_{\text{Minibatch loss: try to fit well to the training data}} + \underbrace{\lambda \Omega(W)}_{\text{Parameter norm penalty: don't fit too well to the training data}}$$

Minibatch loss: try to fit well to the training data

Parameter norm penalty: don’t fit too well to the training data

Dropout: How to train over-parameterized networks?



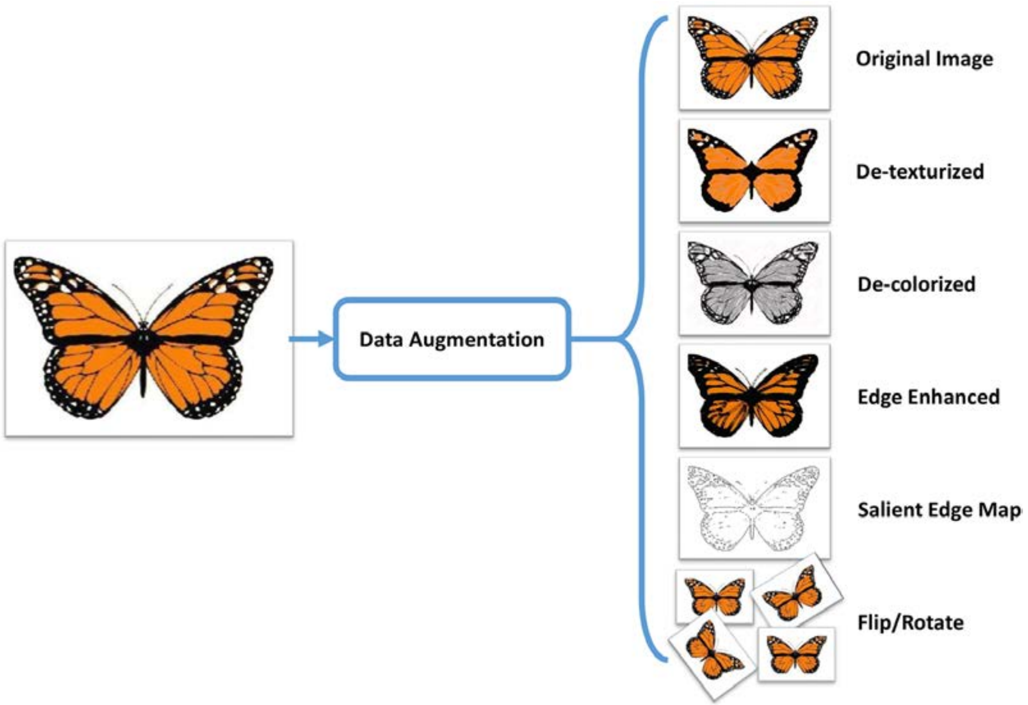


Dropout: randomly dropping out network connections with a fixed probability during training.

Data augmentation

The best approach to improve the performance of your model is to increase the size of the training dataset.

One can also augment the dataset by applying (sometimes random) transformations to the original data. Your task, and thus model, should be invariant to such transformations.

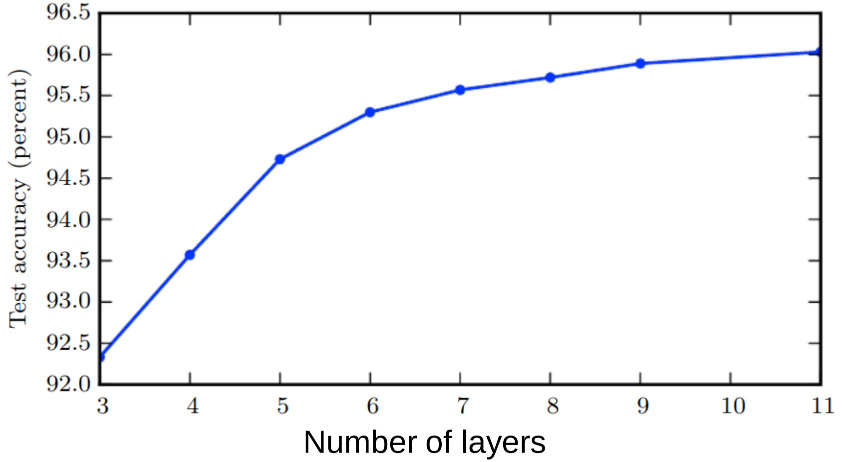


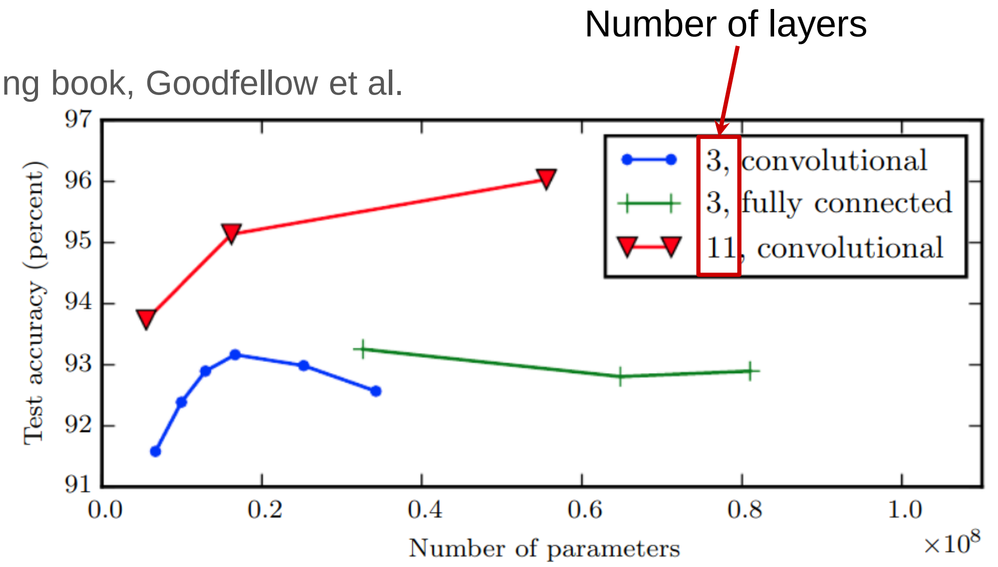
Examples of augmentation transformations, suitable transformations are data and task dependent.

Figure credit: Ahmad, Muhammad, Baik (2017). doi.org/10.1371/journal.pone.0183838

The importance of depth

Goodfellow et al. [arXiv:1312.6082](https://arxiv.org/abs/1312.6082). And Deep Learning book, Goodfellow et al.

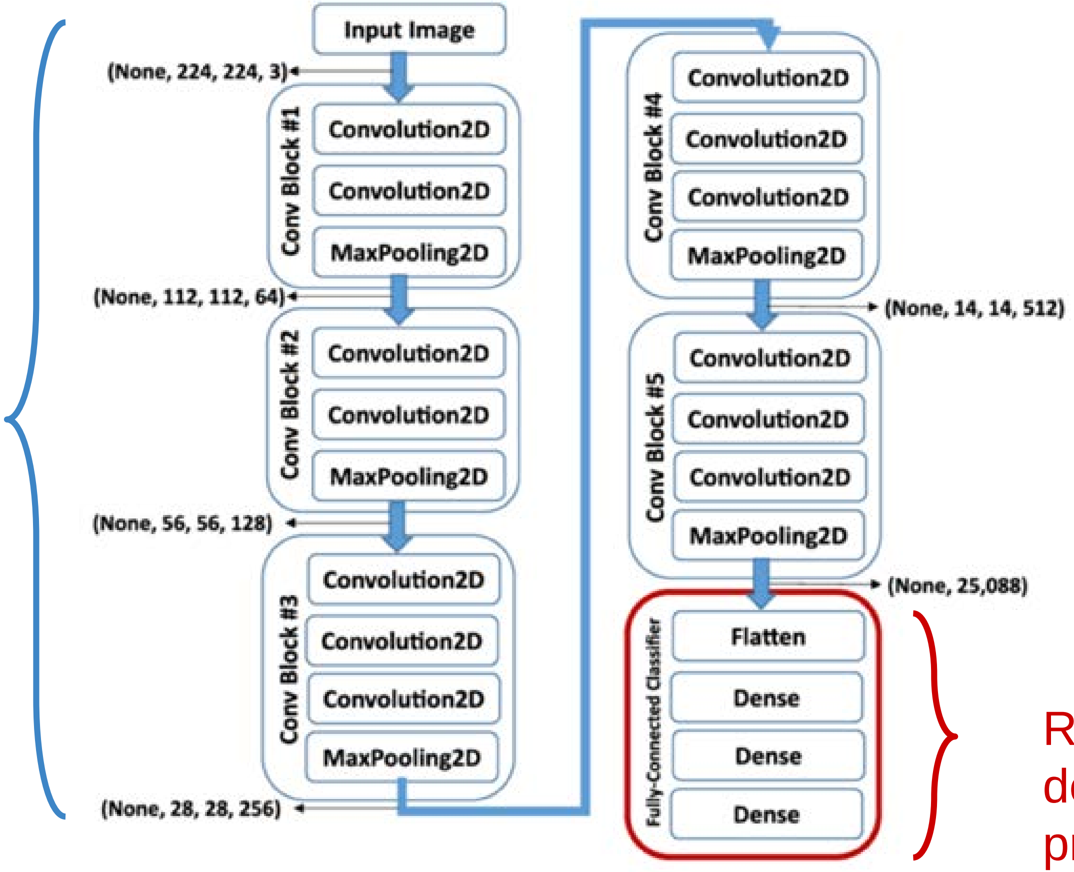




Accuracy of model increases as depth increases.

Deeper models outperform wider modes with the same total number of parameters.

Transfer Learning



Use pre-trained conv layers (feature extractors). You can also fine tune them on your data

Re-train one or more of the dense layers on your problem

Hyper-parameters Optimization (HPO)

Hyperparameters to tune: network architecture, learning rate, its decay schedule, regularization (L^2/dropout strength)

More training tips:

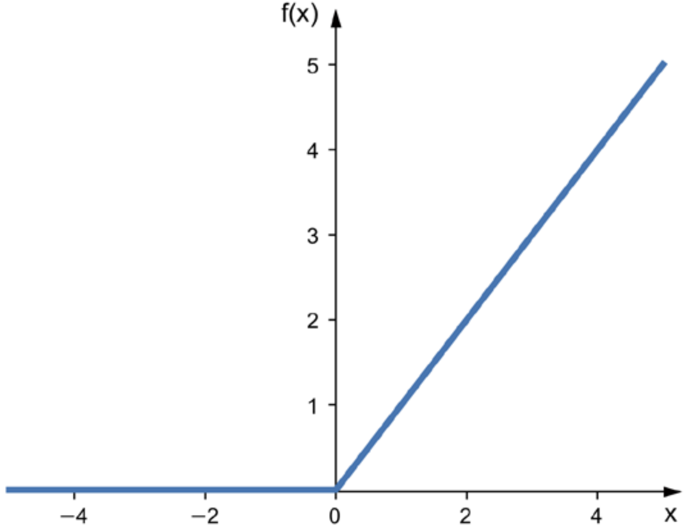
Monitor activations distributions: useful to spot problems with initializations, too many dead activations … etc

Monitor update scales (gradients/weights ratio) should be ~0.001 - 0.01 of weights

EXTRAS

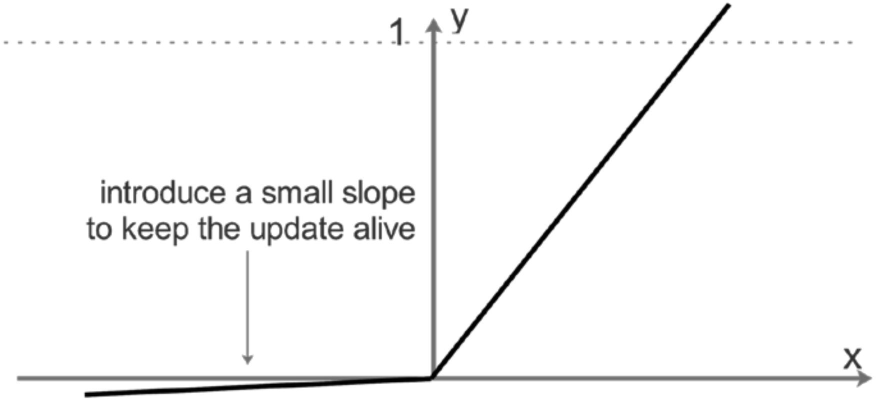
More on activations: Rectified Linear Unit (ReLU)

- Always non-negative
- Computationally cheap
- Passes strong gradients for $x > 0$
- Dies for $x < 0 \rightarrow$ leads to neurons with sparse activity



$$\mathrm{ReLU}(x) = \max(0, x)$$

More on activations: Leaky Rectified Linear Unit (ReLU)

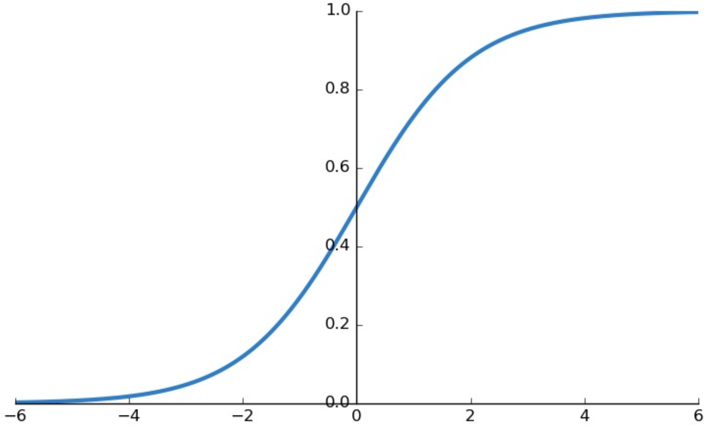


$$\text{Leaky ReLU}(x) = \max(\alpha x, x)$$
$$0 < \alpha < 1$$

Fig credit: wangxinliu.com/machine%20learning/machine%20learning%20basic/research&study/NN1/

More on activations: Sigmoid

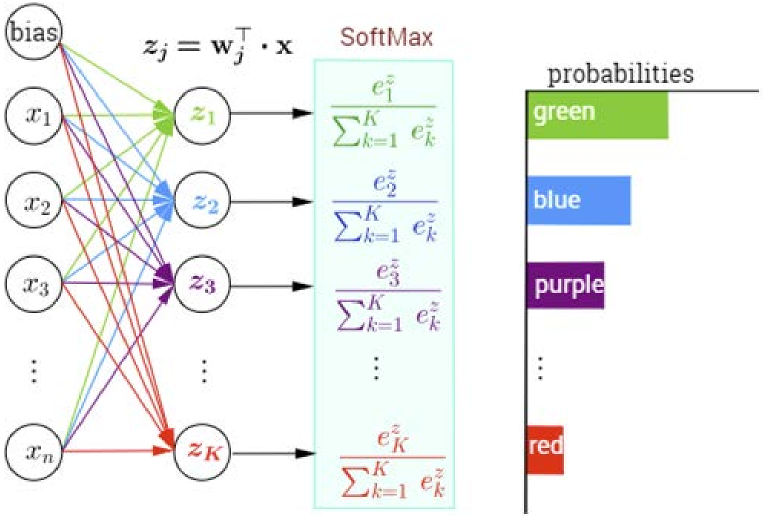
- Bounded between 0 and 1
- Useful to squash layer output to represent binary probability → Bernoulli output distribution
- Expensive to compute
- Saturates at low and high input values → small slopes → low gradient signal → needs a **Log** in the loss function to cancel the effect of the **Exp**



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

More on activations: Softmax

- Multinoulli output distribution → multi-class output
- Produces a distribution over classes
- Predicted class is the one with the largest probability
- Needs a **Log** in the loss function to cancel the **Exp**



$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$