Neural Network Tools and Principles, part 1

Building blocks of computational intelligence

Computational Intelligence in Games, Spring 2018
Prof. Perttu Hämäläinen
Aalto University

Contents

- **Preliminaries: compute graphs, artificial neurons, activation functions, loss functions**
- Understanding nonlinear activations
- Understanding skip-connections
- Convolutional neural networks
- Encoder-decoder architectures

3Blue1Brown YouTube channel: Visual intuitions to math and neural networks

- https://www.youtube.com/watch?v=aircAruvnKk (what is a neural network)
- https://www.youtube.com/watch?v=IHZwWFHWa-w (how neural networks learn, i.e., gradient descend)
- https://www.youtube.com/watch?v=Ilg3gGewQ5U (what is backpropagation really doing)
- Essence of Linear Algebra (vectors, matrices, determinants etc., the branch of math at the heart of it all): https://www.youtube.com/watch?v=kjBOesZCoqc&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab

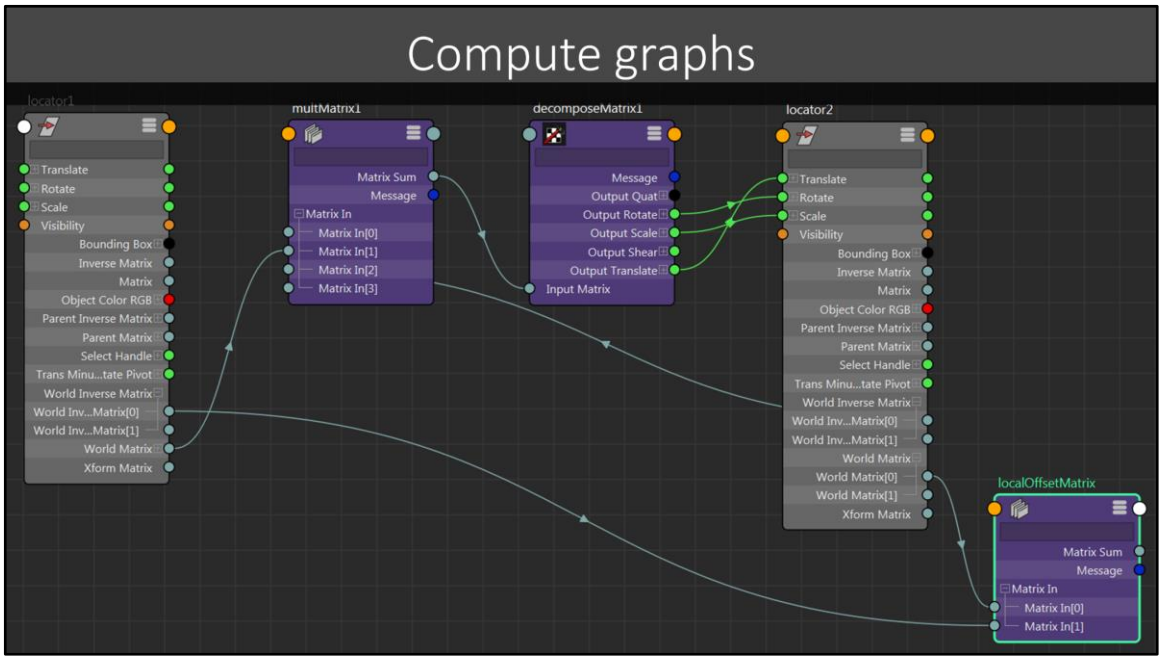
Each video is around 10-20 minutes. Watch at least the first one.

We have these already in the course prerequisites. If you skipped them, please watch ASAP. The linear algebra ones are optional, but included in case you want to feel bad about how high-school math used to be taught when you were in high school :) I for sure would have loved this material when I was studying.

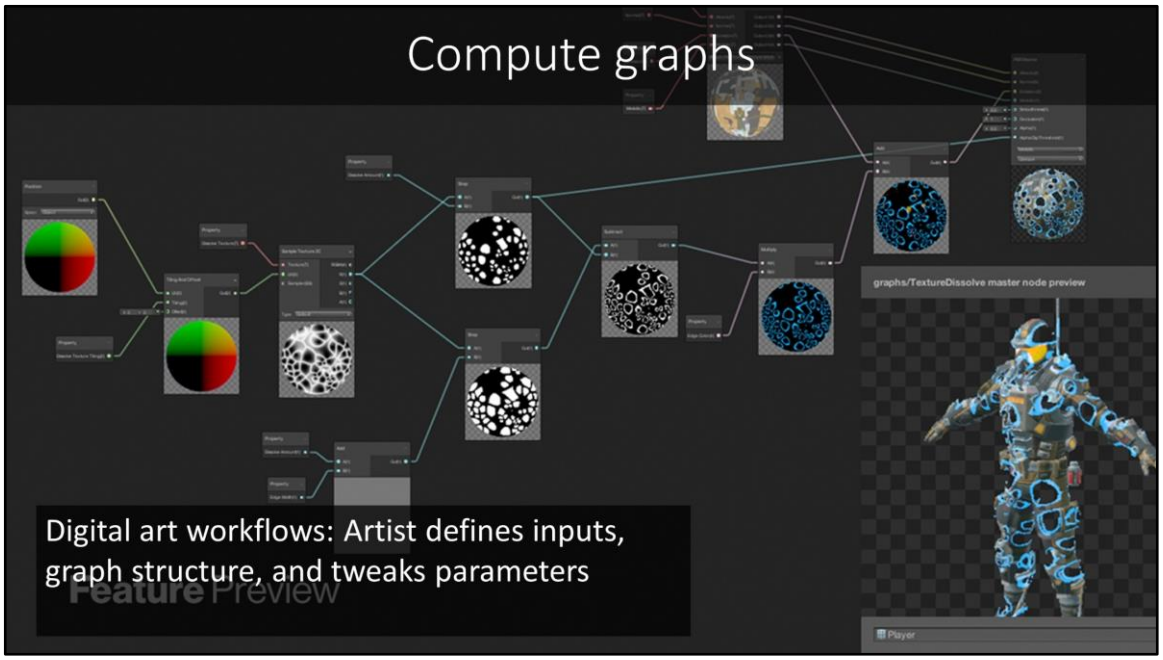
The videos, including all animations, arefully coded in Python, and the python source code of all the channel's videos are on Github:

https://github.com/3b1b/manim

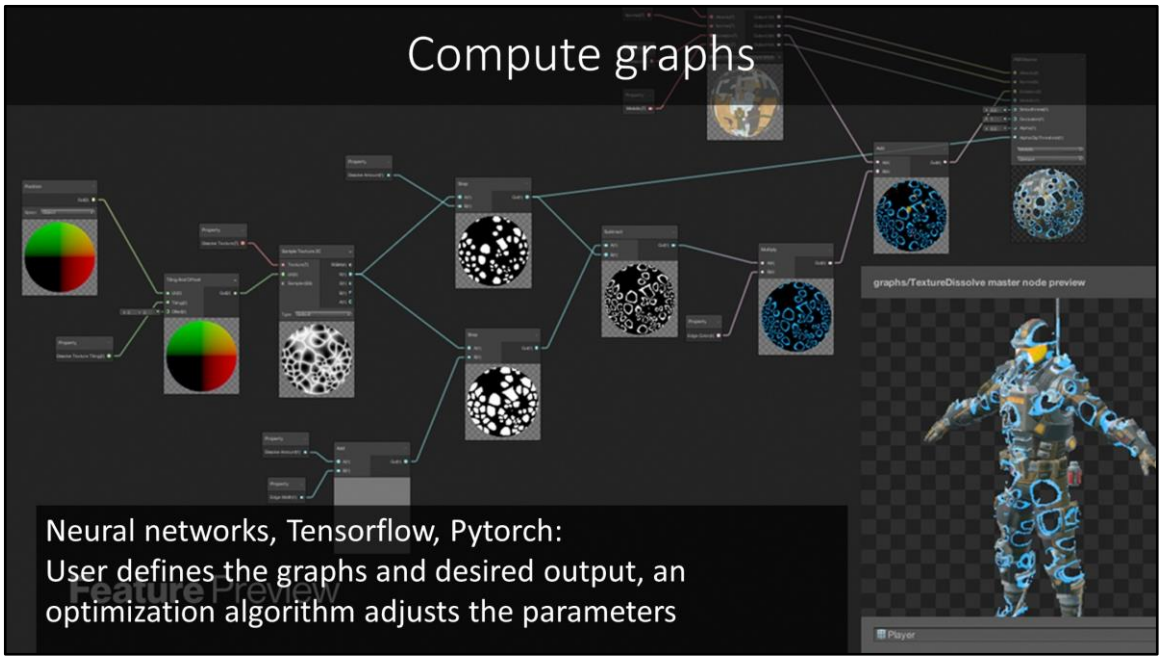
Compute graphs



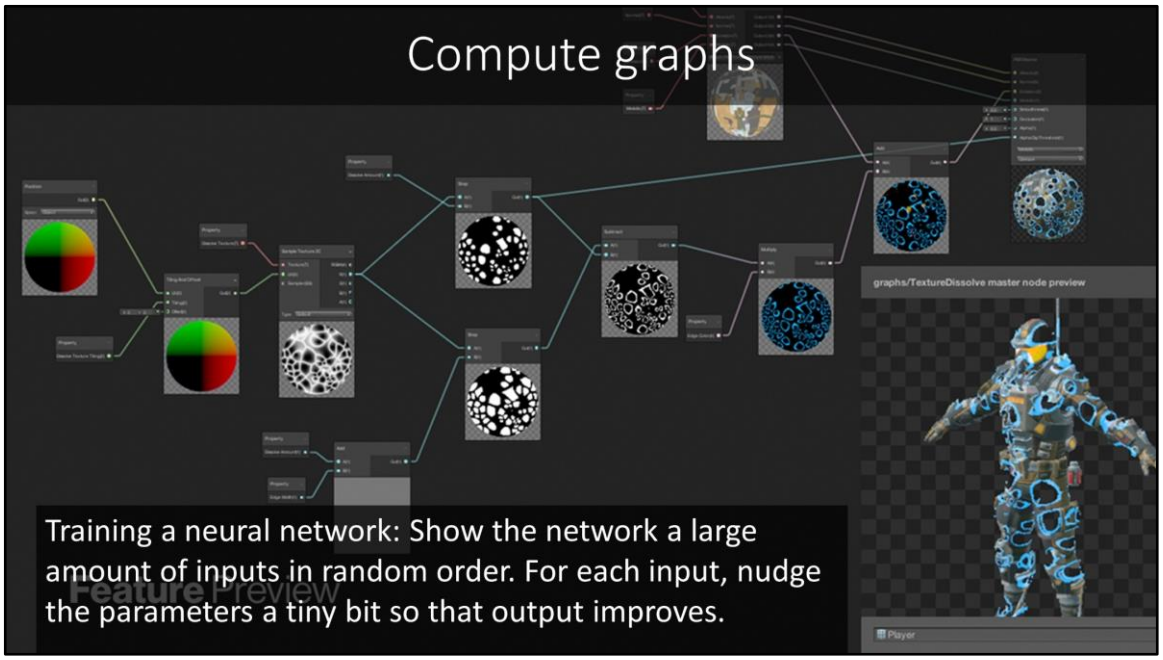
Maya's node editor is an example of a compute graph: Operations through which data flows.



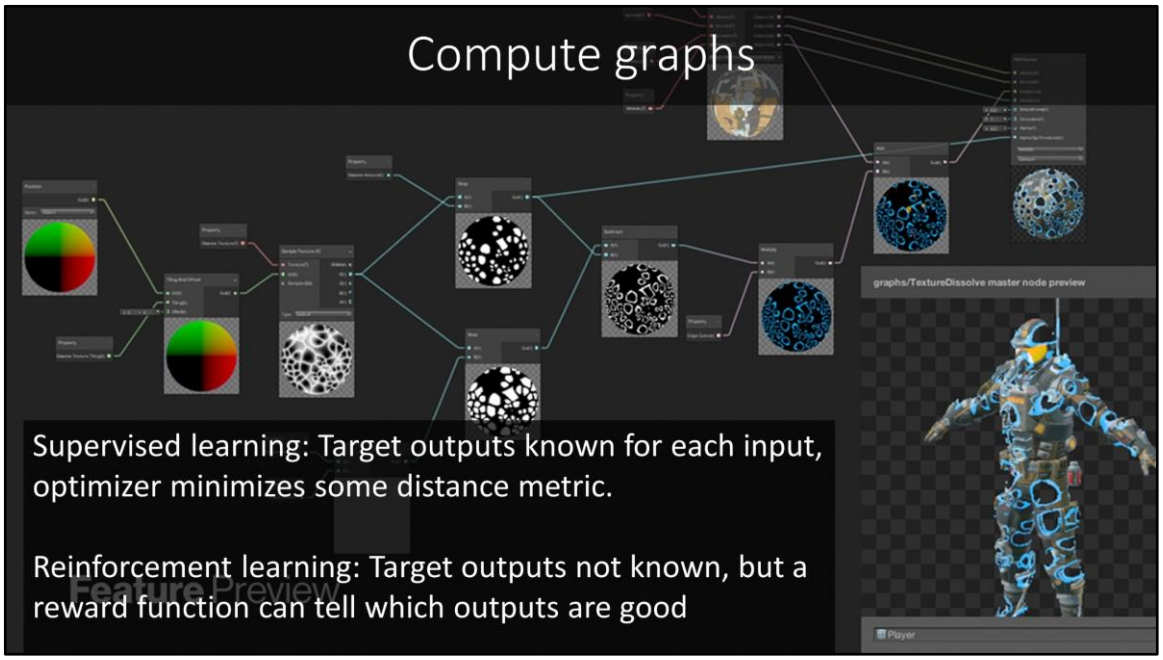
Unity shader graph



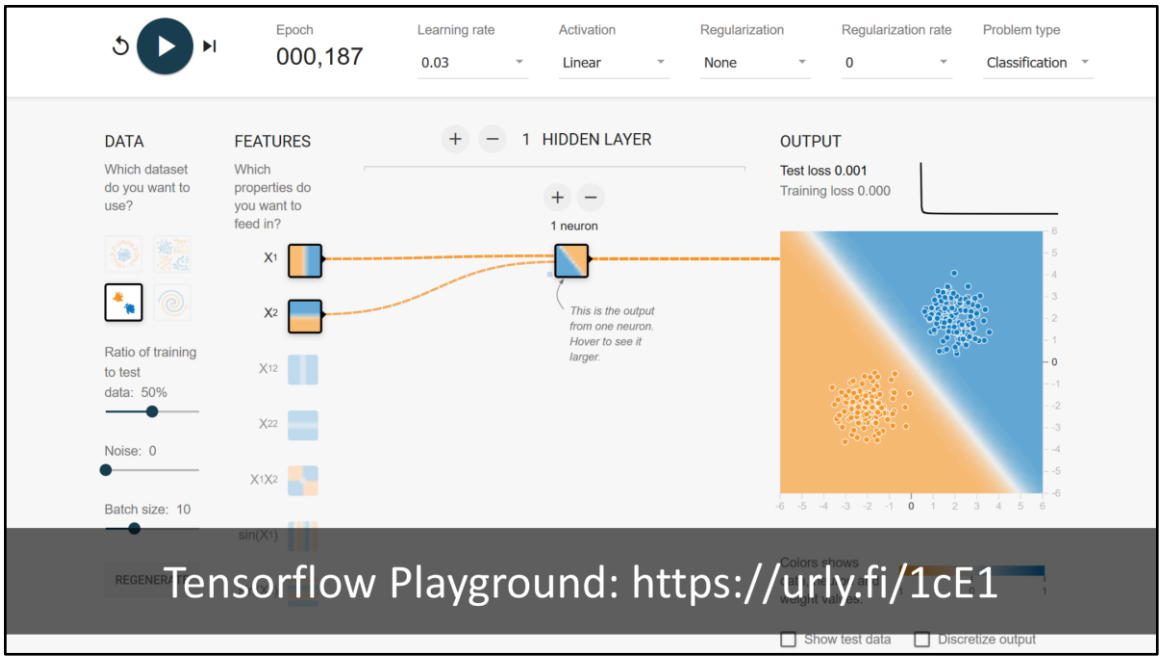
Unity shader graph



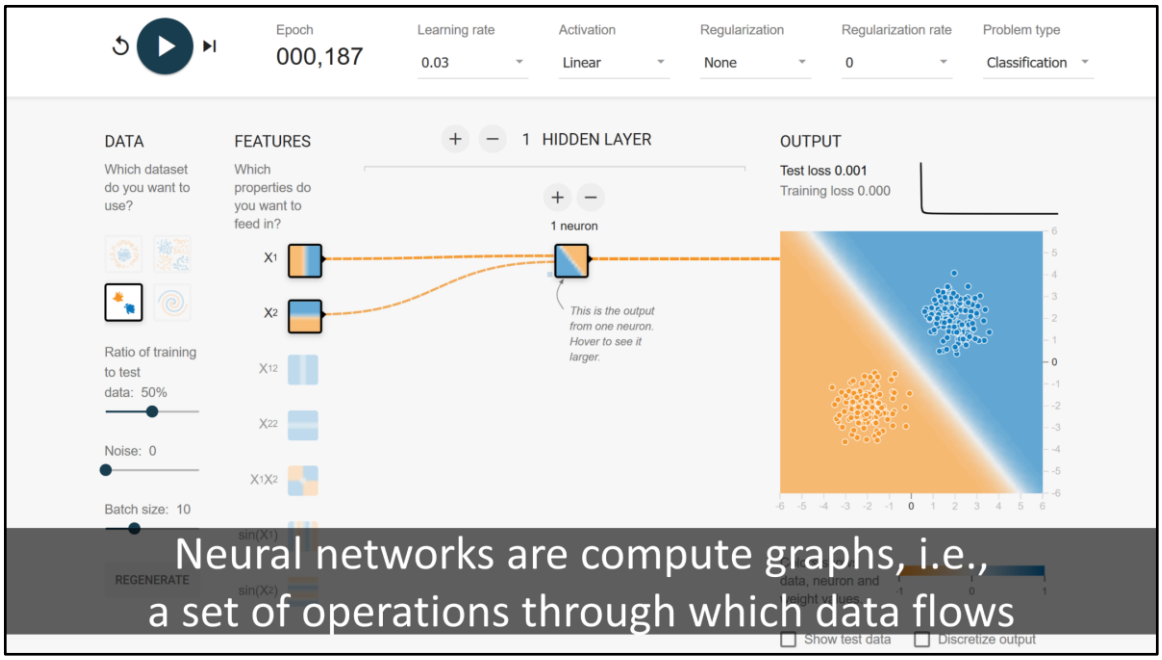
Unity shader graph

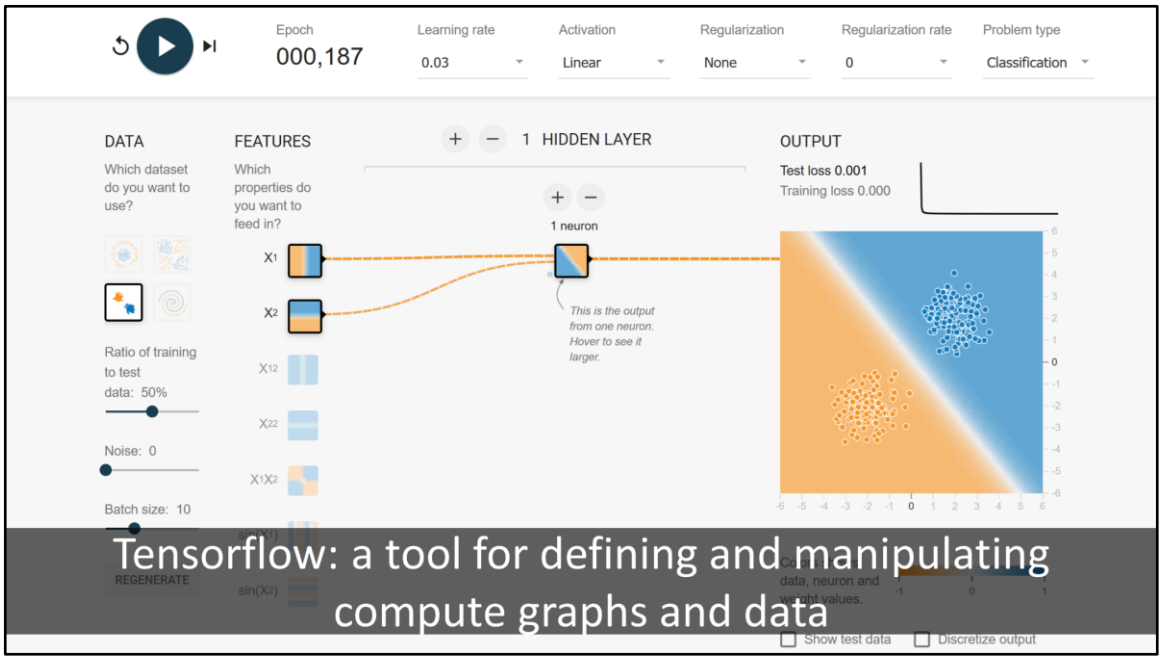


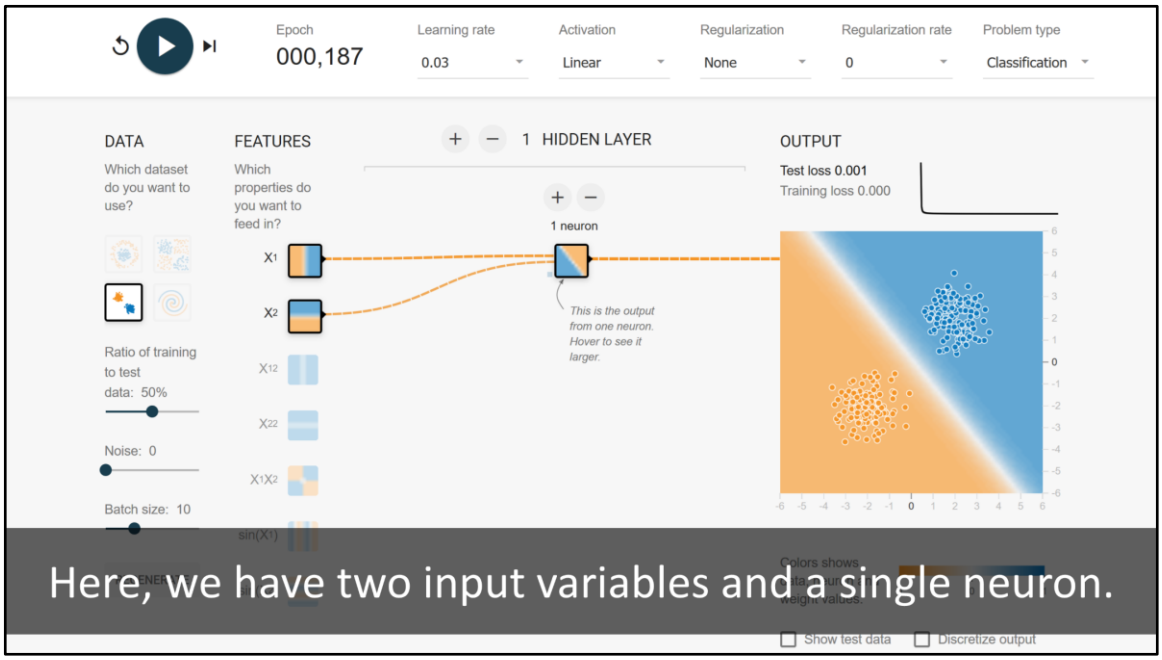
For example, when the graph outputs game actions, the system can get a reward for winning the game, and no reward for losing.



Open this URL


Epoch
000,187
Learning rate
0.03
Activation
Linear
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X1)
sin(X2)
1 HIDDEN LAYER
1 neuron
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.001
Training loss 0.000
Show test data
Discretize output
Neural networks are compute graphs, i.e.,
a set of operations through which data flows


Epoch
000,187
Learning rate
0.03
Activation
Linear
Regularization
None
Regularization rate
0
Problem type
Classification
DATA
Which dataset do you want to use?
Ratio of training to test data: 50%
Noise: 0
Batch size: 10
REGENERATE
FEATURES
Which properties do you want to feed in?
X1
X2
X12
X22
X1X2
sin(X2)
1 HIDDEN LAYER
1 neuron
This is the output from one neuron. Hover to see it larger.
OUTPUT
Test loss 0.001
Training loss 0.000
Show test data
Discretize output
Tensorflow: a tool for defining and manipulating compute graphs and data

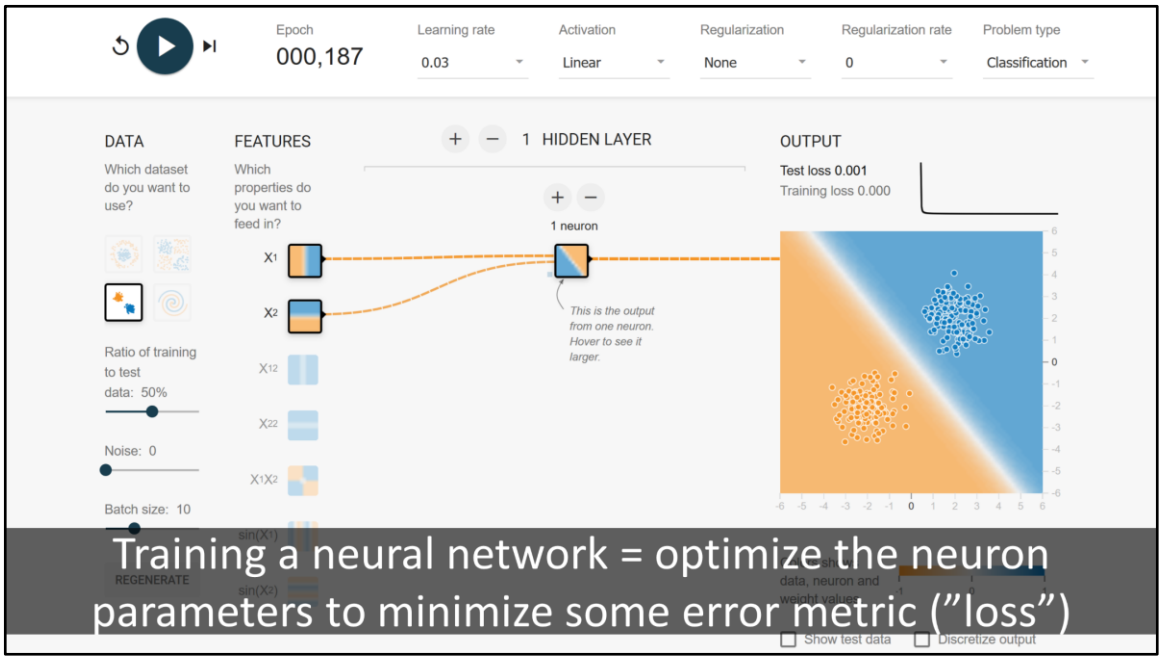


x1=horizontal axis, x2=vertical axis

The neuron tries to predict the color (orange or blue) of each input x1,x2 pair.

On the right, we see both the data points and the color prediction of the network in the 2D space of x1,x2 coordinates.

Key takeaway: a single neuron can split the data linearly into two classes.



Parameters = the ”synaptic weights” of a neuron.

In the image, the error metric is the training loss reported at top-right corner.

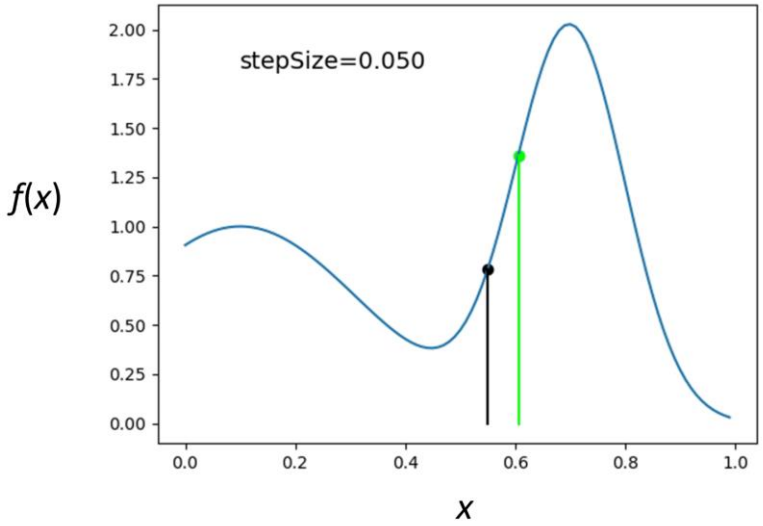
Try optimizing the 1-neuron network in Tensorflow Playground, just hit play at the top left corner

All AI is (mathematical) optimization

- Adjust some **x** to minimize or maximize some $f(\mathbf{x})$
- Usually, one denotes vectors with boldface and scalars with italic, i.e., $\mathbf{x}=[x_1,x_2,...]$
- Neural network training: **x** denotes network parameters, $f(\mathbf{x})$ is the loss or error function
- Pathfinding: **x** denotes path steps, $f(\mathbf{x})$ is path length
- Gameplay AI: **x** denotes actions, $f(\mathbf{x})$ is the utility or cost function
- Animation: **x** denotes muscle activations or other movement synthesis parameters, $f(\mathbf{x})$ measures goal attainment, e.g., based on player input.
- Neural networks: optimal **x** has to be found through numerical iteration. (No closed-form solution that could be derived algebraically)

We will get back to this later, but good to remember already now.

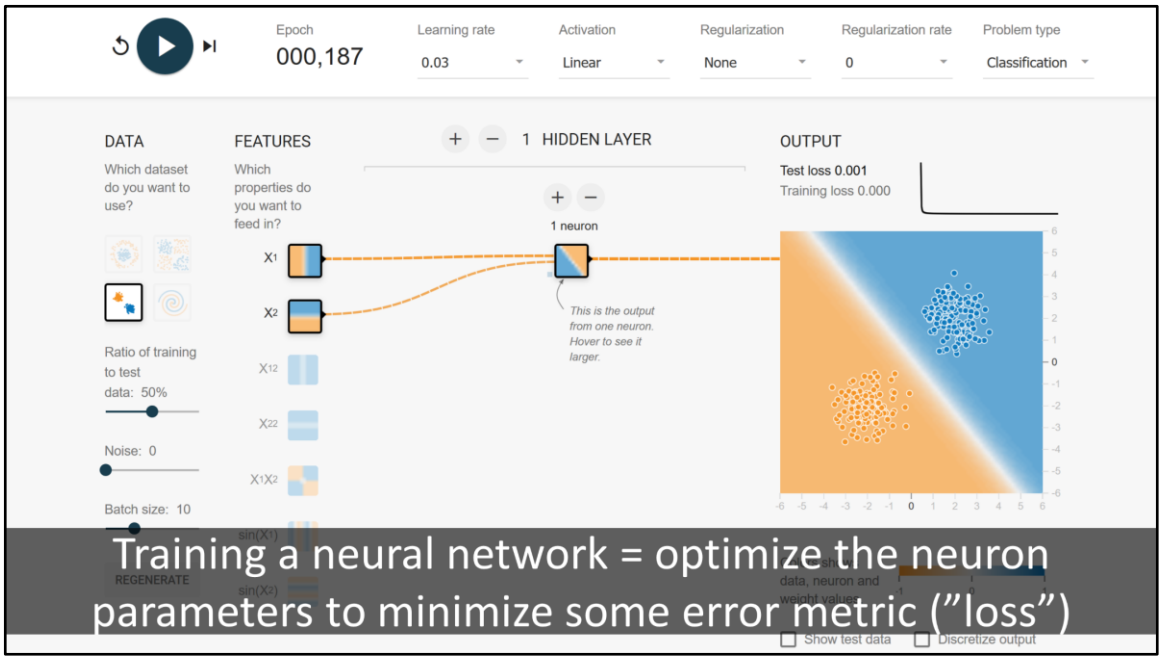
Optimization = exploring the *f*(x) landscape



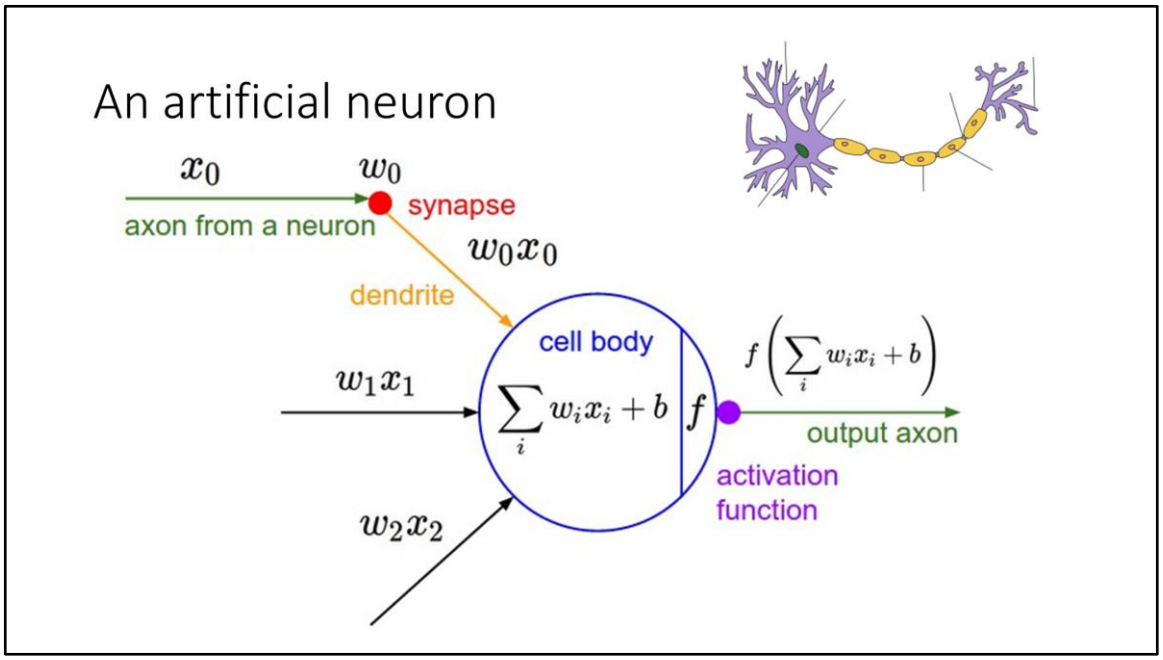
Here's an example of a one-dimensional *x* and some *f*(*x*). Various optimization algorithms implement different strategies of climbing upwards or downwards in the *f*(*x*) "landscape".

In this animation, we simply try random new *x* around the current *x* and accept if the new *f*(*x*) is better (shown in green).

In neural network optimization, one typically computes the gradient of *f*(*x*), which tells in which direction *f*(*x*) grows fastest.



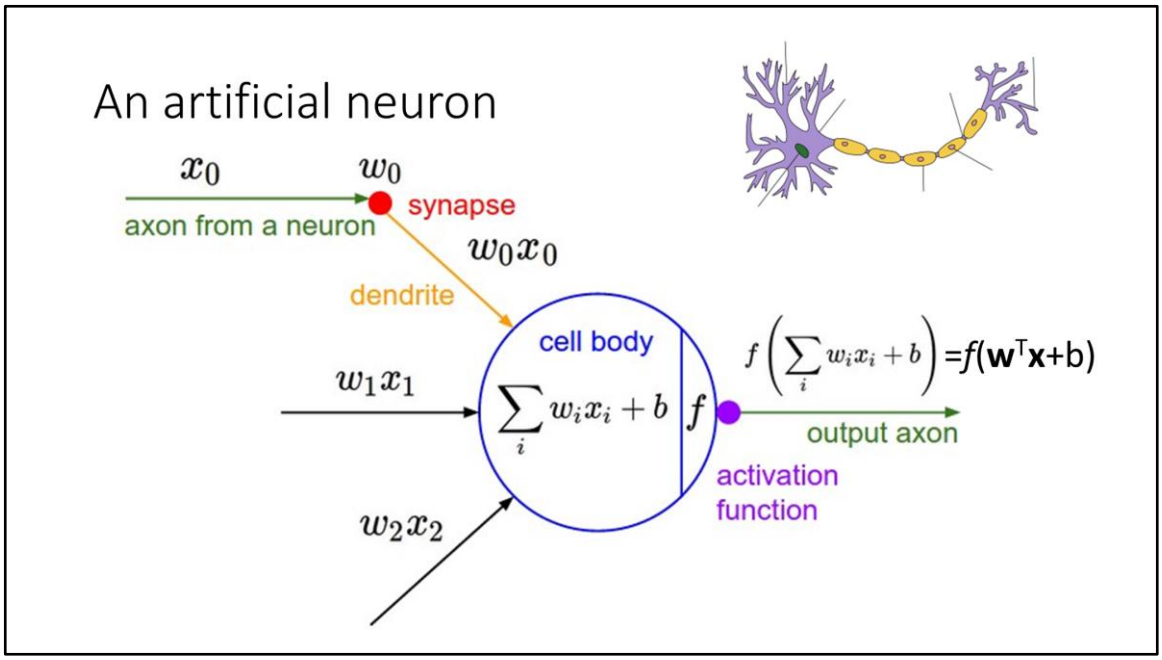
Again, here we optimize the parameters of a neuron. So what are these parameters?



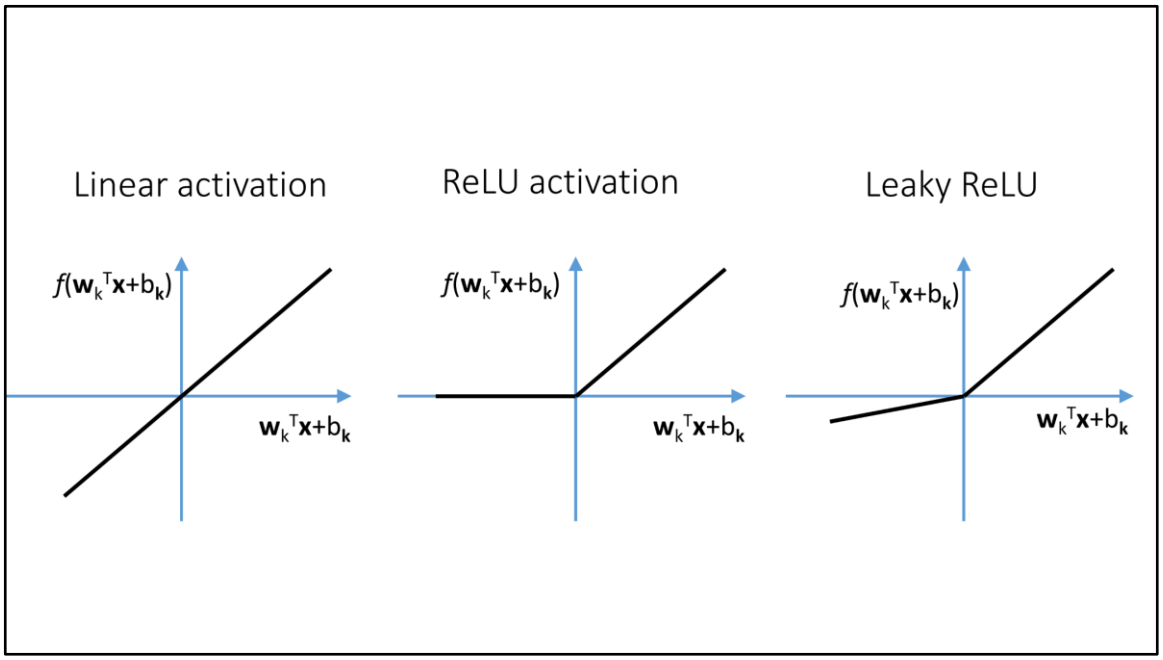
A neuron is just a mathematical operation: sum together all the inputs x_i weighted by the synaptic weights w_i, then add a bias b and apply an activation function f(x)

Each neuron outputs the result, i.e., a single number. In a neural network, there are typically many neurons organized as layers, which means that each layer outputs a vector of these numbers. They then become the inputs of the next layer.

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



We usually use vector & matrix notation to get rid of the subindices. The $\mathbf{w}^T\mathbf{x}$ denotes the dot product operation, i.e., multiplying together the elements of vectors **w** and **x** and then summing the results.



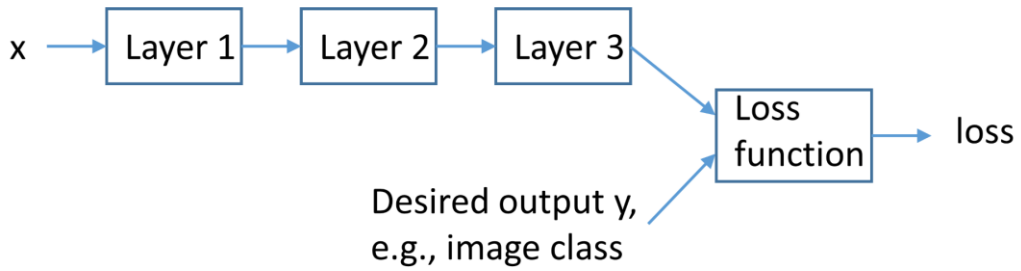
In addition to **w** and b, each neuron has the activation function f().

Linear activation = no additional activation function, *f(x)=x.* Historically, this was the case with early models.

ReLU (Rectified Linear Unit) and Leaky ReLU are among the most common activation functions nowadays.

Training = optimization, minimizing loss

1. Feed a random *minibatch* of input data x through the network
2. Compute the loss function for each pair
3. Change the weights such that loss decreases, on average
4. Rinse & repeat!



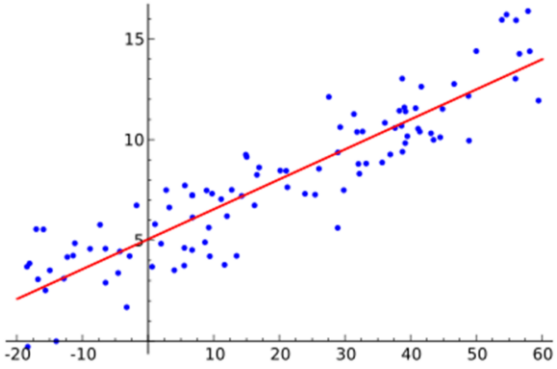
Typically, the datasets are so large that we don't have the memory or computing time to feed all data through the network. However, in practice we can just take a random subset of data.

Since we only have a minibatch of the data, we cannot even theoretically adjust the parameters to the optimal value at once. Thus, we instead nudge them a bit towards a direction that minimizes the loss for the minibatch. With smaller minibatches, training is faster, but the direction estimates become more noisy.

When repeated many times with small enough nudges and large enough minibatches, the minibatch sampling noise averages out and does not break the training. In fact, the noise can even help the optimization escape so-called saddle points, which we will discuss later in more detail.

Common loss functions

- Sum of squared errors, i.e., differences between network output variables and desired output variables



This is used in regression tasks, and produces optimal results if one assumes the data is corrupted by Gaussian noise. Here, x axis is the input variable, y is the output variable, red line is the neural network's output, and blue are noisy training samples.

Image source: https://en.wikipedia.org/wiki/Linear_regression

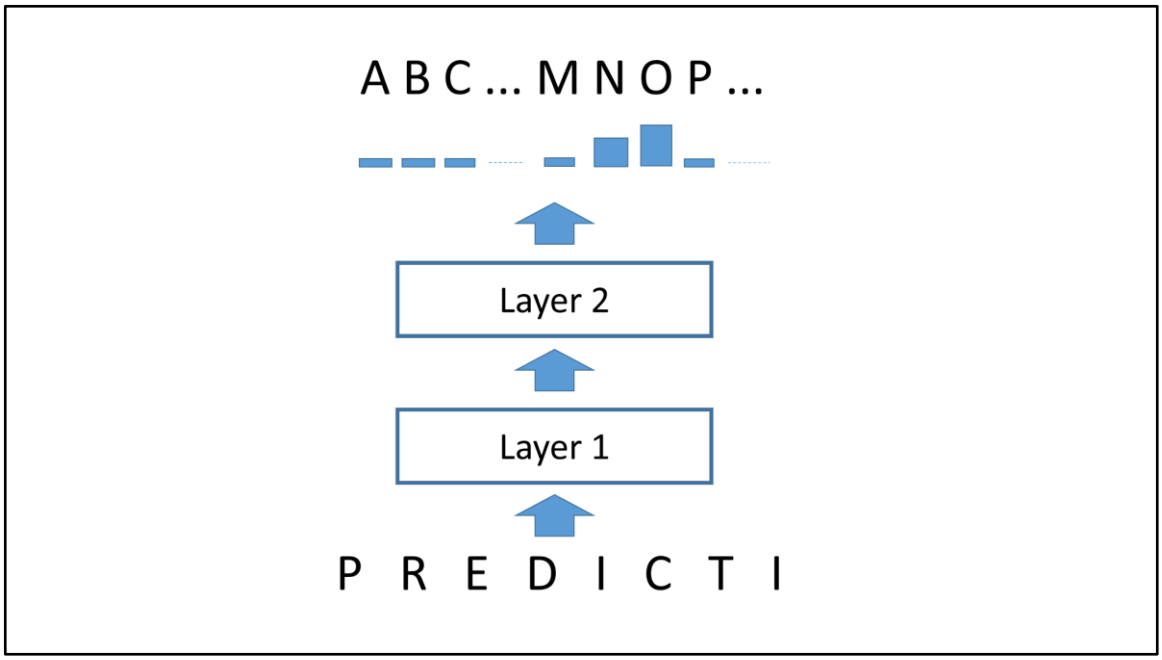
Common loss functions

- Softmax cross-entropy, in classification where one output neuron for each class

```
def cross_entropy(X,y):
    """
    X is the output from fully connected layer (num_examples x num_classes)
    y is labels (num_examples x 1)
    """
    m = y.shape[0]
    p = softmax(X)
    log_likelihood = -np.log(p[range(m),y])
    loss = np.sum(log_likelihood) / m
    return loss
```

More more, see: https://deepnotes.io/softmax-crossentropy

Classification tasks perform much worse with squared error loss. Note: the last layer neuron activations are first converted into a discrete probability distribution (each output in range 0...1 such that they sum to 1) using the softmax function. The last two lines then compute the cross-entropy, a measure of differences between probability distributions. The cross-entropy is taken with respect to the target probability distribution *y*, usually encoded such that the correct class probability is 1 and all the others are 0.



With softmax output layer and cross-entropy loss, the network learns to predict the probabilities for different outputs.

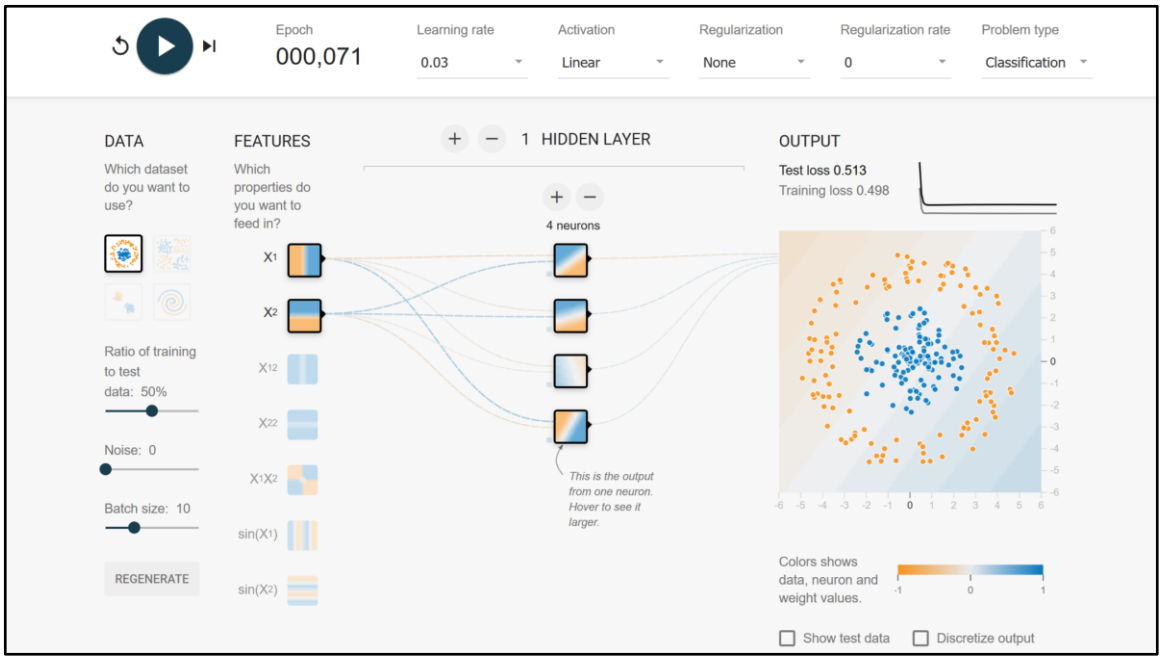
Simple example: predict the next character, or the probabilities of possible next characters.

This could be trained with the basic tools we've covered so far: just have a standard classification network with an output neuron for each possible character, and train with softmax cross-entropy loss, using random 8-character input snippets from some text data.

This can then be used in two ways: pick the most probable character, or randomly sample a character based on the probabilities. Now, if one shifts the input sequence to the left and adds the sampled character to the right, one can use this kind of a network for text generation.

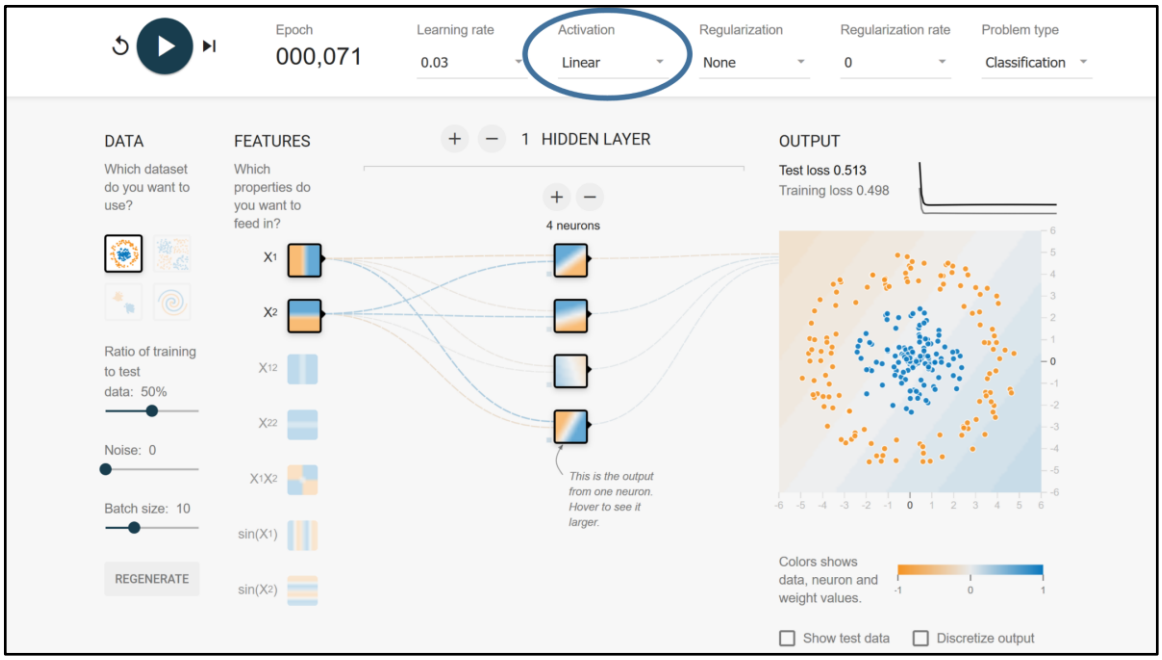
Contents

- Preliminaries: compute graphs, artificial neurons, activation functions, loss functions
- **Understanding nonlinear activations**
- Understanding skip-connections
- Convolutional neural networks
- Encoder-decoder architectures

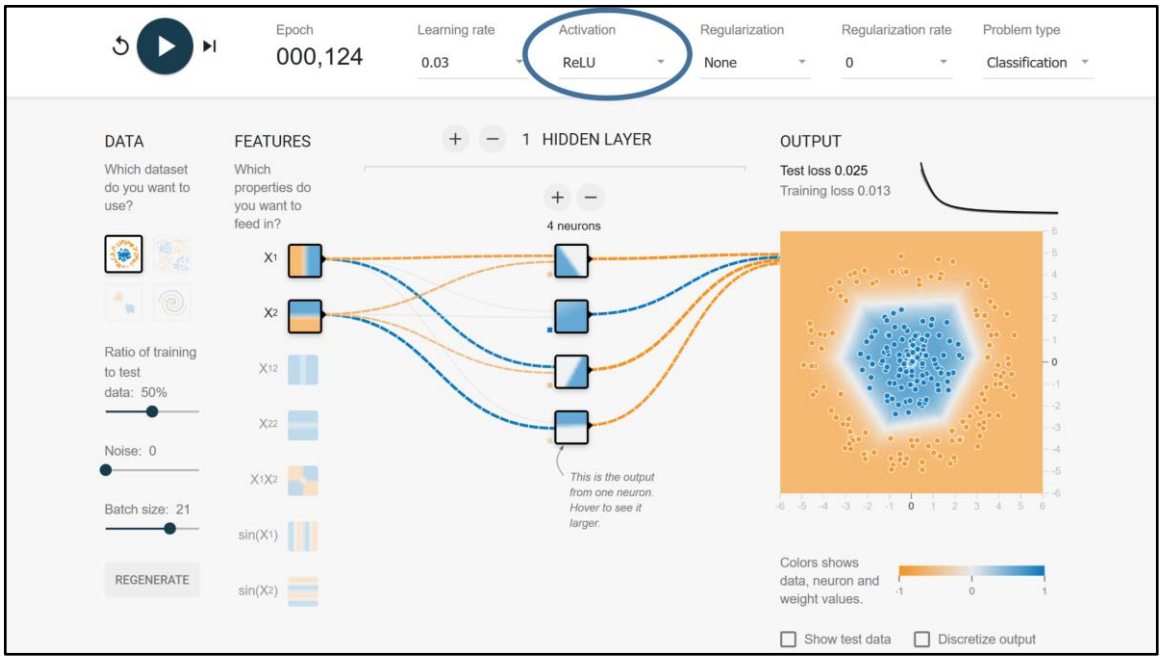


This classification toy example uses softmax cross-entropy loss.

If the problem is more difficult, one usually needs to add neurons. So why do we not get better results here?

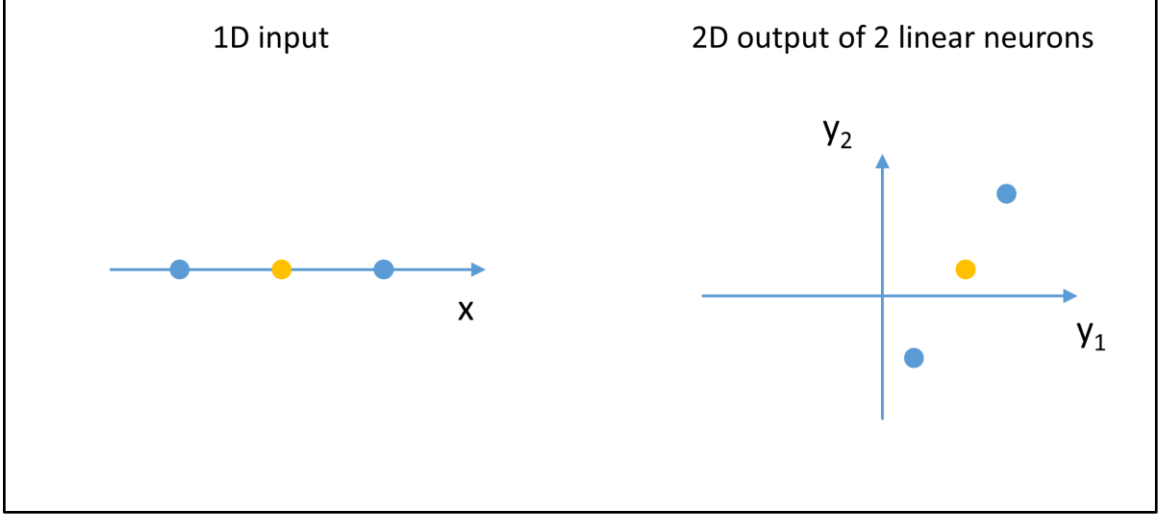


The key is that with a linear activation function, many neurons essentially behave the same as a single one.



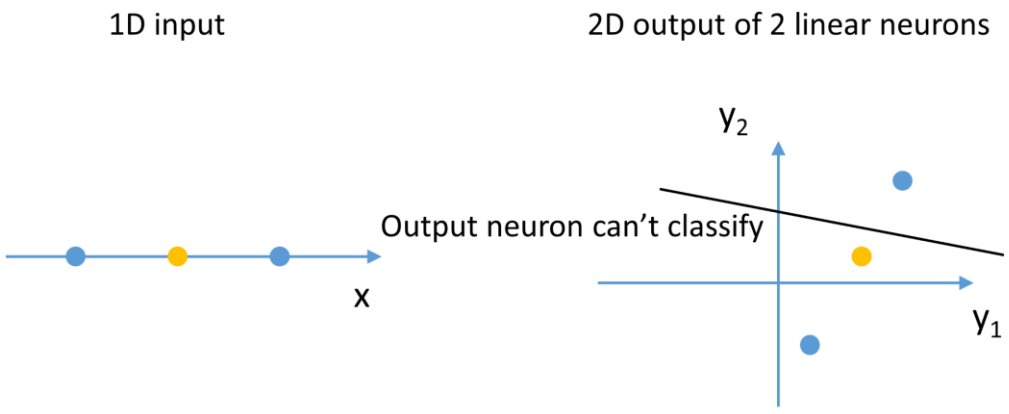
Simply switching to relu activation allows solving this simple classification problem.

The effect of nonlinearity

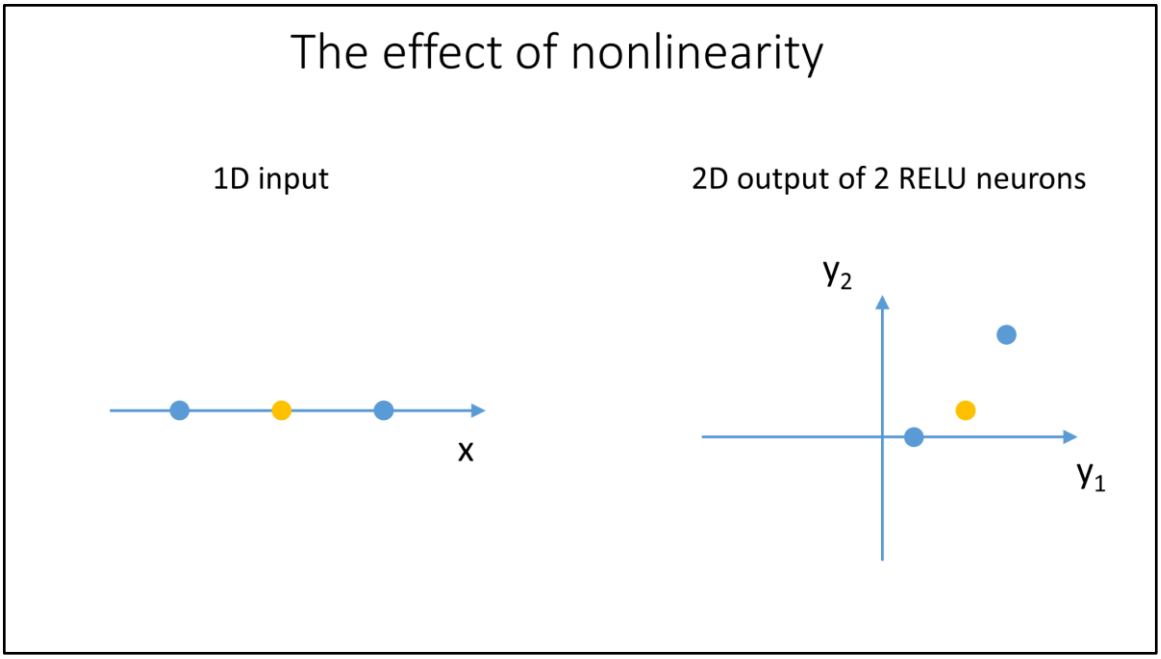


Let's visualize why this is so

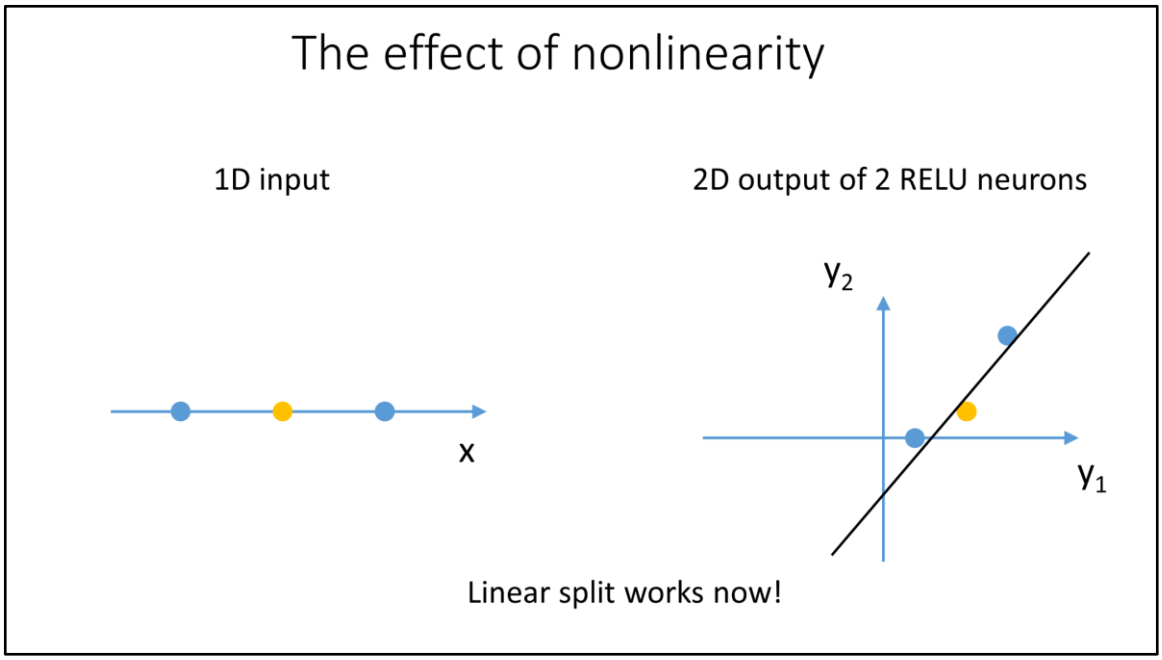
The effect of nonlinearity



The effect of nonlinearity

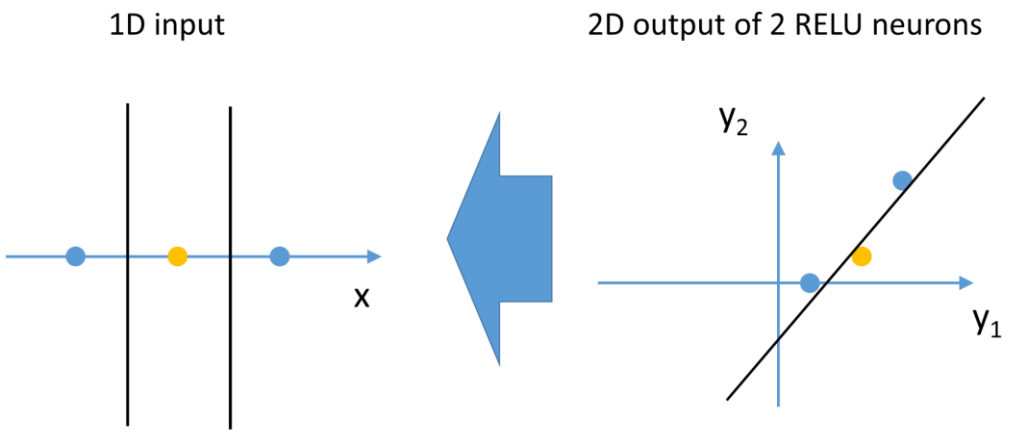


k denotes neuron index

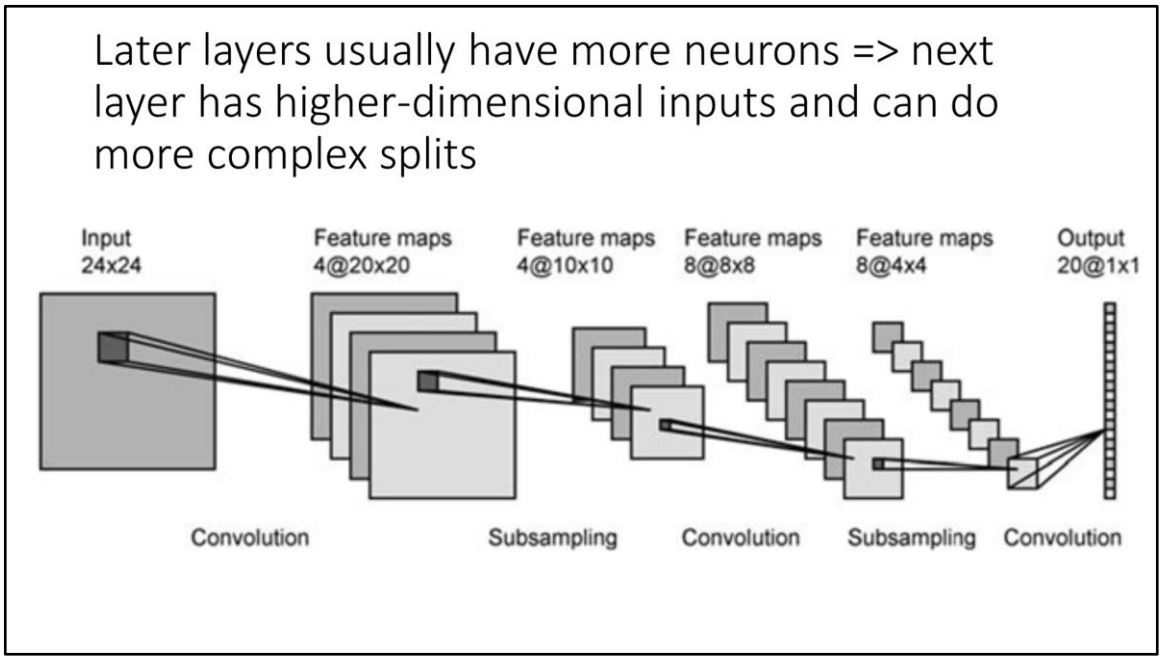


This process can then be continued by further layers, combining and dividing the previous layers' outputs in novel ways.

Split into two halves in a high-dimensional representation = multiple splits in the input space

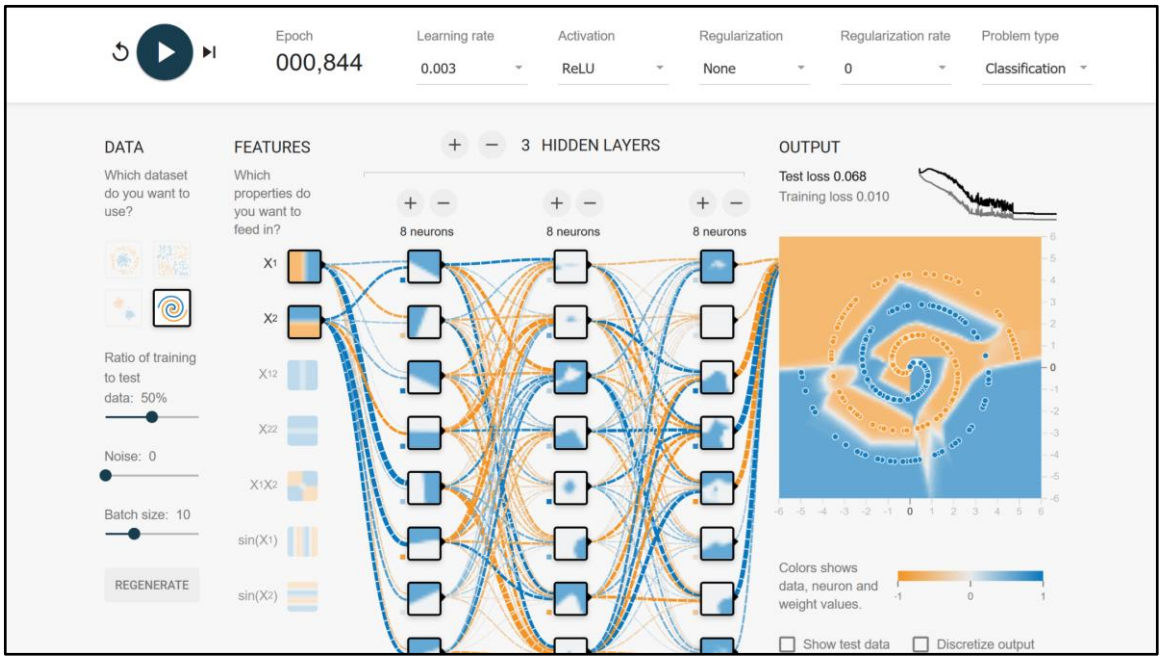


This process can then be continued by further layers, combining and dividing the previous layers' outputs in novel ways.



This is a typical image processing neural network. Each square represents the output of one neuron, more about that later. The main point is that in many real-world architectures, the number of neurons doubles for each layer or grows in some other manner. This increase of dimensionality basically allows each neuron to carve out more complex regions in the input space. This is the key principle for a neuron to recognize images or parts of images such as noses, eyes or mouths – the network must learn to separate the recognized objects in the original input space of image pixels, which is very high-dimensional (24x24=576 in the image, each pixel brightness is treated as a separate input variable)

Deep learning refers to networks that have many layers. Some say that anything beyond 3 layers is deep.



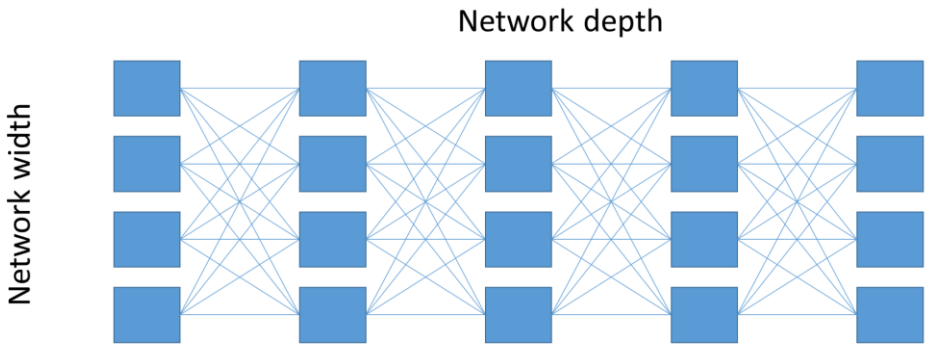
Tensorflow playground visualizes these regions or decision boundaries for each neuron in the input space, see the small rectangles. Note how they grow more complex at further layers. The first layer neurons only do linear splits, while the subsequent neurons can carve out both concave and convex regions.

The output layer always has a single neuron, linear in this case (followed by thresholding to allow using this for classification)

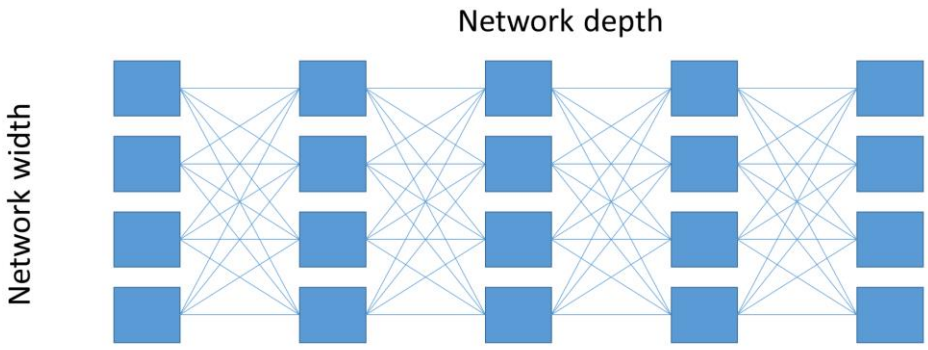
At least in some cases, it has been shown that the amount of regions a network can distinguish grows **exponentially** with the number of layers. This is a mindblowing result; the difficulty of dealing with exponentially complex problems is what plagues other machine learning methods such as shallow neural networks, decision trees or forests, and K nearest neighbors.

Adding neurons to a single hidden layer only increases model resolution linearly. Thus, one gets much better return on investment by adding layers.

Intuition on the power of depth



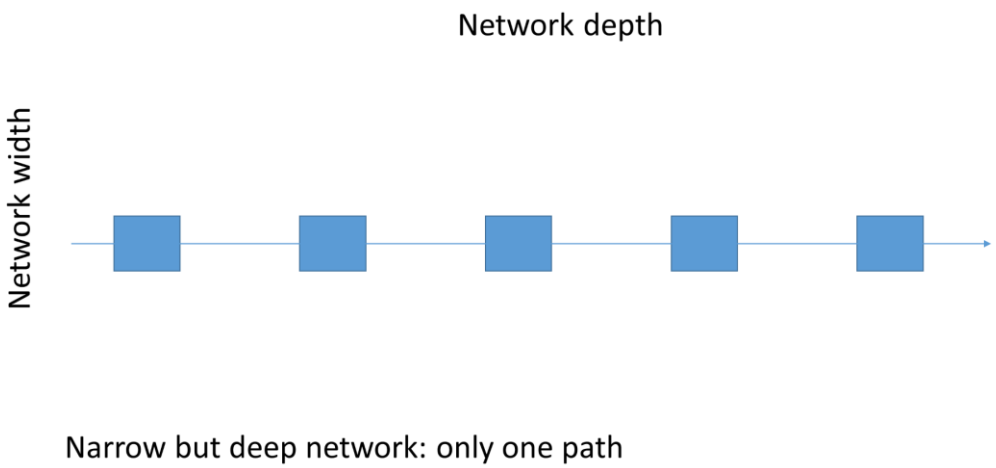
Intuition on the power of depth



Neurons turning on and off ”route” the data through the network along different paths.

How many paths can the data take through the network?

Intuition on the power of depth

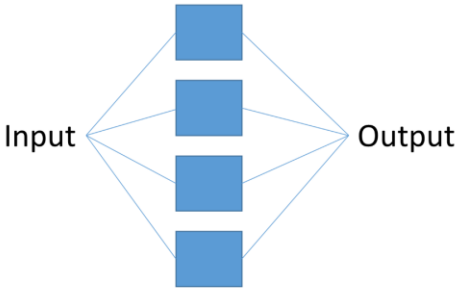


Narrow but deep network: only one path

Also: since each neuron only outputs a single variable, it cannot represent complex data.

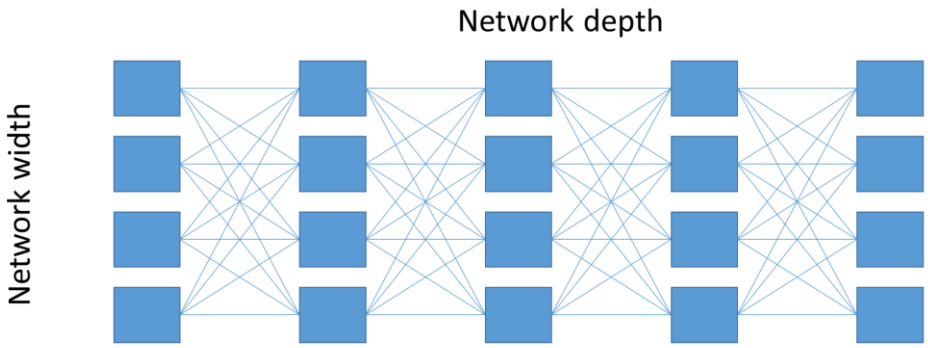
Network width = dimensionality of the data passing through the network. Can't compress 100x100 pixel images (10000-dimensional data) down to 1D.

Intuition on the power of depth



Wide but shallow network: Only as many paths as there are neurons

Intuition on the power of depth



Wide and deep network: $\text{width}^{\text{depth}}$ paths. This is the power of deep learning.

In early machine learning, we just didn't know how to optimize the networks in a way to make the power of depth emerge.

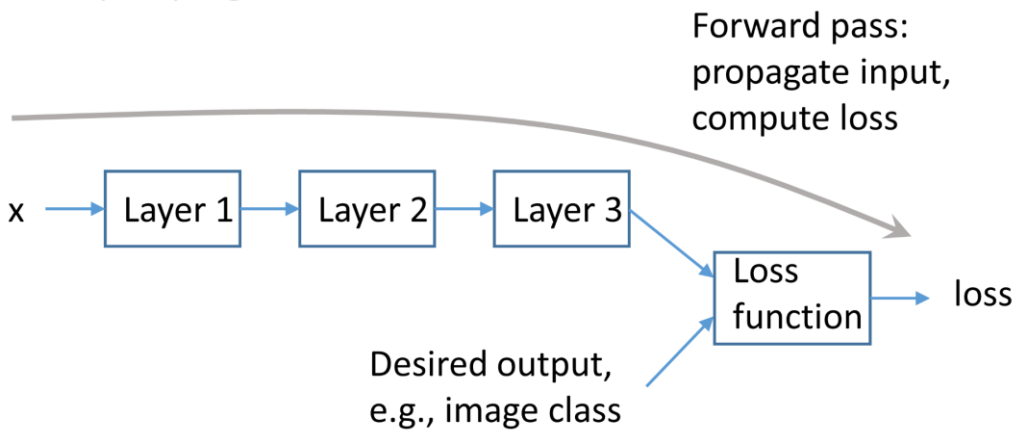
Contents

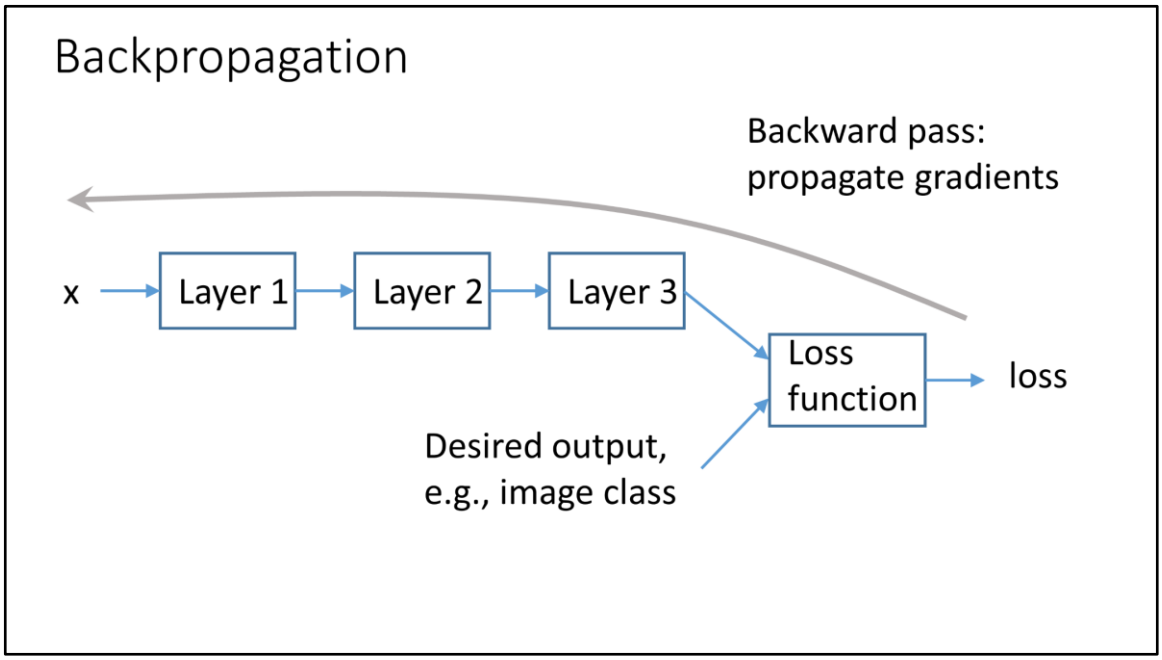
- Preliminaries: compute graphs, artificial neurons, activation functions, loss functions
- Understanding nonlinear activations
- **Understanding skip-connections**
- Convolutional neural networks
- Encoder-decoder architectures

Problems with adding layers

- In practice, optimization uses gradients (more about that in the optimization lecture!)
- Gradients may vanish & explode with deep networks => optimization does not converge
- This is because gradient computation multiplies together matrices from all the layers (backpropagation, i.e., chain rule of calculus)

Backpropagation



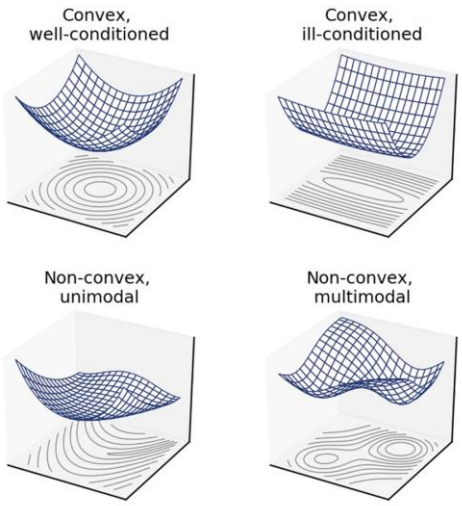


The gradients of the minimized loss w.r.t. each layer's parameters are used in the training optimization. At each block of the graph,matrices are multiplied together to compute, e.g., layer 2 gradients based on already computed layer 3 gradients. This can be considered a dynamic programming procedure, i.e., breaking down a big problem into subproblems (here, following the graph structure), and calculating the solution recursively based on the subproblem solutions.

Compute graphs and gradients

- Key: every operation is differentiable, i.e., can compute gradient of output w.r.t. params or input
- Gradient = which direction to nudge parameters to improve the loss function?
- *Tensorflow etc. are tools for building compute graphs out of operations that can automatically compute their gradients*
- Optimization depends on gradients because the high number of parameters makes random and/or global search for the optimum infeasible

Types of optimization problems



The key insight here is that when a problem is convex and well-conditioned (top-left corner), gradient of $f(\mathbf{x})$ points to towards the optimum and optimization is fast. In other cases, gradient-based optimization can take costly detours and/or get stuck before reaching the true optimum. The black curves show the path taken by gradient descend.

Convex: the isocontours of $f(\mathbf{x})$, shown in gray in the figures, are convex, as opposed to concave.

Well-conditioned: the isocontours are spherical as opposed to elongated. Mathematically, the Hessian of $f(\mathbf{x})$ is close to an identity matrix multiplied with a scalar, having a low ”condition number”, i.e., the ratio of largest and lowest eigenvalues.

In higher-dimensional problems, you can think in terms of isosurfaces instead of isocontours.

Solutions to gradient problems

- SELU activation function (self-normalizing signals to zero-mean unit variance)
- Skip-connections

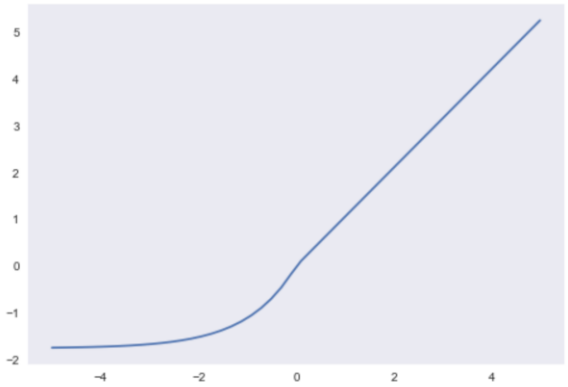


Image: SELU activation. Like ReLU, but with a smoother nonlinearity. Gives very good results, allows training very deep networks even without skip-connections. However, not common outside fully connected networks.

Self-Normalizing Neural Networks
Günter Klambauer, Thomas Unterthiner, Andreas Mayr, Sepp Hochreiter
https://arxiv.org/abs/1706.02515

Skip-connections: the ResNet

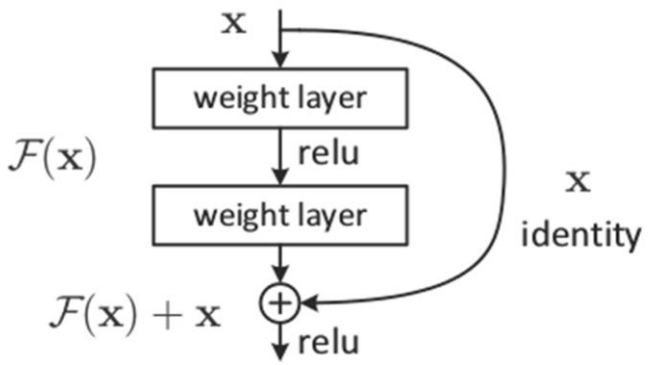


Figure 2. Residual learning: a building block.

Many modern architectures include some form of skip-connections, which let the gradients ”flow” over layers in backpropagation without the stacked multiplication.

In effect, this lets the optimization make the network behave like a shallow network when possible (easy to optimize), and utilize the power of depth only when needed (harder to optimize).

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

Abstract

Deeper neural networks are more difficult to train. We present a residual learning framework to ease the training of networks that are substantially deeper than those used previously. We explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. We provide comprehensive empirical evidence showing that these residual networks are easier to optimize, and can gain accuracy from considerably increased depth. On the ImageNet dataset we evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [40] but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test *set. This result won the 1st place on the ILSVRC 2015 classification task. We also present analysis on CIFAR-10 with 100 and 1000 layers.*

The depth of representations is of central importance

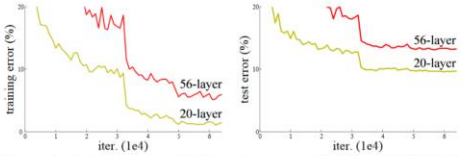


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

greatly benefited from very deep models.

Driven by the significance of depth, a question arises: *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients [14, 1, 8], which hamper convergence from the beginning. This problem,

This is the paper that introduced residual learning with skip-connections.

For a later analysis, see: Veit, Andreas, Michael J. Wilber, and Serge Belongie. "Residual networks behave like ensembles of relatively shallow networks." *Advances in Neural Information Processing Systems*. 2016.

A related later technique: Huang, Gao, et al. "Densely connected convolutional networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. Vol. 1. No. 2. 2017.

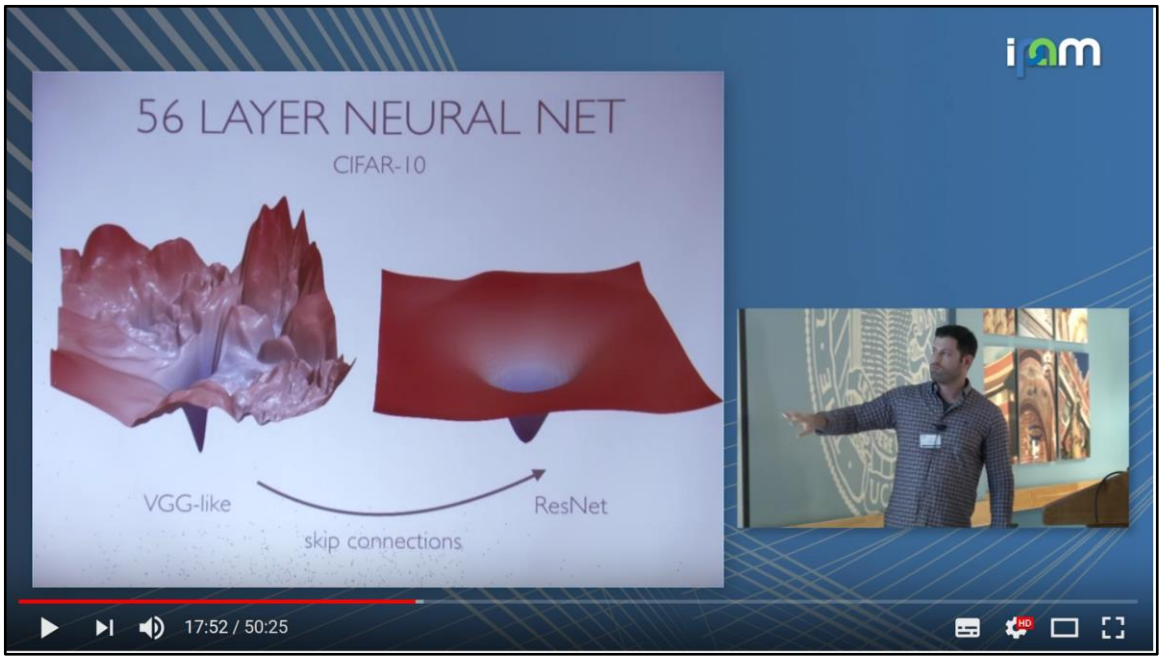


Image from this awesome talk by Tom Goldstein: https://www.youtube.com/watch?v=78vq6kgsTa8 (”What do neural loss surfaces look like?”)

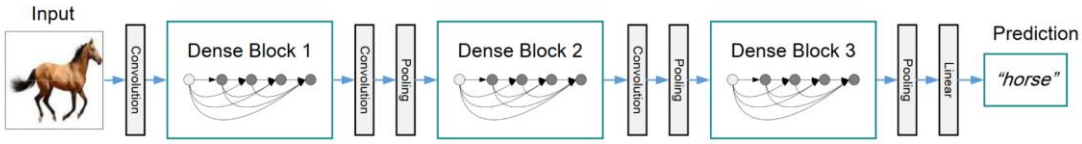
VGG-like: no skip-connections.

CIFAR-10: an image recognition benchmark.

Adding skip-connections makes the optimization much more smooth, which allows gradient descend to converge faster and more robustly. In the landscape without skip-connections, optimization will get stuck if and when the initialization is not within the narrow basin around the optimum.

Note that these surfaces are plotted as 2D slices of the full space of network parameters, which has millions of dimensions, but the intuition they give is very much in line with empirical results.

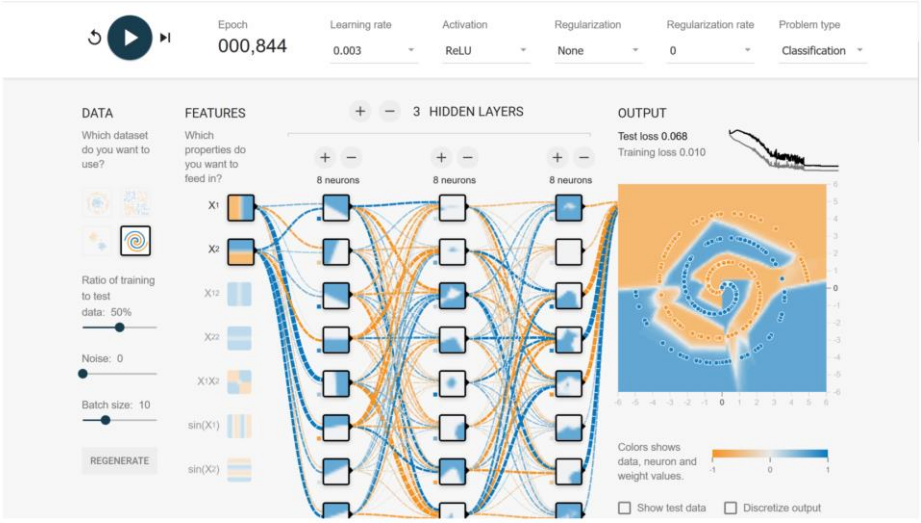
The DenseNet



A related later technique: Huang, Gao, et al. "Densely connected convolutional networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. Vol. 1. No. 2. 2017.

Here, “dense” refers to skip-connections. In each dense block, the amount of incoming skip-connections increases at each layer, which means that the neurons operate on high-dimensional spaces where they can make more complex decisions (carve out more complex regions of the original input space).

This is deprecated for deep networks



In summary, in most real-world cases, people don't anymore use this kind of basic multilayer networks. Better results are usually obtained with ResNets, DenseNets, Convolutional Neural Networks, or other more recent architectures.

If one for some reason wants to use a network without skip-connections, one should at least consider using SELU activations.

Contents

- Preliminaries: compute graphs, artificial neurons, activation functions, loss functions
- Understanding nonlinear activations
- Understanding skip-connections
- **Convolutional neural networks**
- Encoder-decoder architectures

Fully connected networks are often inoptimal

- Lots of parameters, slow to optimize
- Assumes that all variables are interrelated, or at least assumes that all interrelations are equally probable
- In real data, relations are often local, e.g., nearby pixels have stronger relationships
- Convolutional networks to the resque! (less parameters, assume locality)

Convolution

- Image blurring or edge detection is ”convolution” with a fixed filter
- Pixel *i* output = $\mathbf{w}^T\mathbf{x}_i$, where $\mathbf{w}$ is the filter kernel and $\mathbf{x}_i$ is an input image patch around the pixel. Filter kernel = neuron parameters
- A convolutional neural network learns such filters that help in minimizing the loss function.

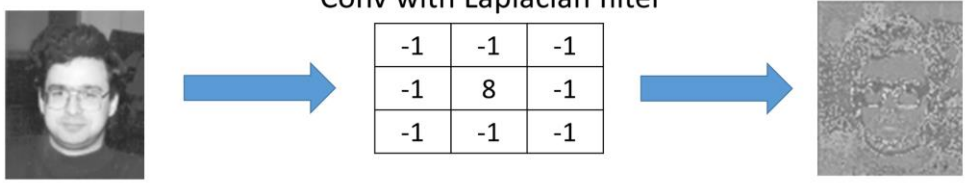
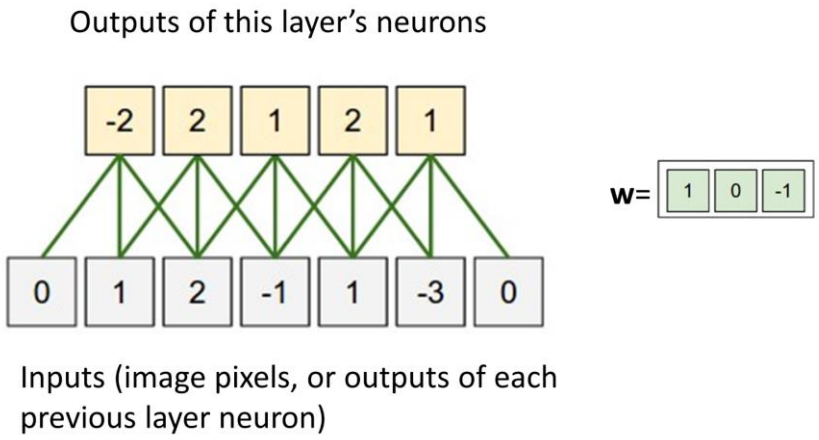


Image source: https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

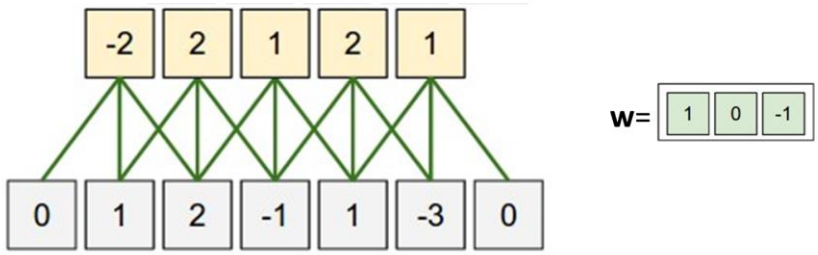
Convolution as a hierarchy, trees



Here, stride=1, i.e., the input patches for each output pixel overlap, which is the default when one applies filters in, e.g., Photoshop. Here, resolution is not decreased to half, and one typically adds a pooling operation.

Convolution as a hierarchy, trees

Convolutional networks: The neurons for each output pixel have the same weights. Implementation: a single filter sweeps over the input array

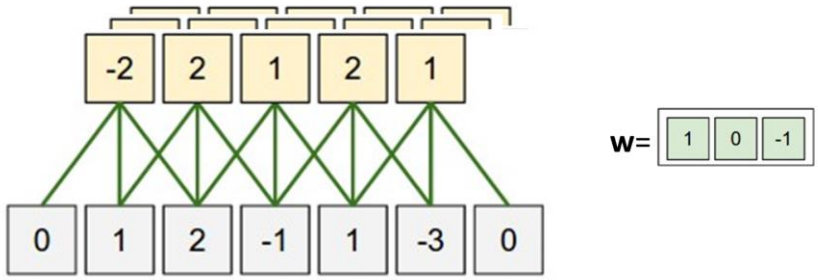


Inputs (image pixels, or outputs of each previous layer neuron)

Here, stride=1, i.e., the input patches for each output pixel overlap, which is the default when one applies filters in, e.g., Photoshop. Here, resolution is not decreased to half, and one typically adds a pooling operation.

Convolution as a hierarchy, trees

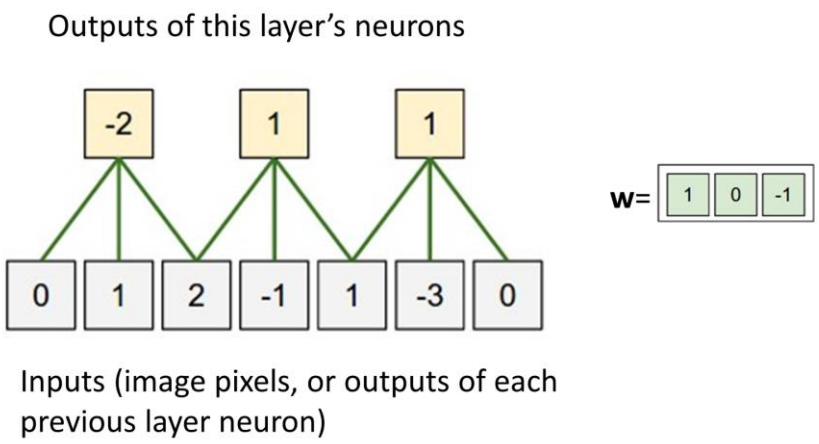
The same is repeated many times, producing multiple ”output channels”. Each channel has its own filter **w**



Inputs (image pixels, or outputs of each previous layer neuron)

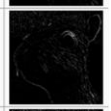
Here, stride=1, i.e., the input patches for each output pixel overlap, which is the default when one applies filters in, e.g., Photoshop. Here, resolution is not decreased to half, and one typically adds a pooling operation.

Convolution as a hierarchy, trees



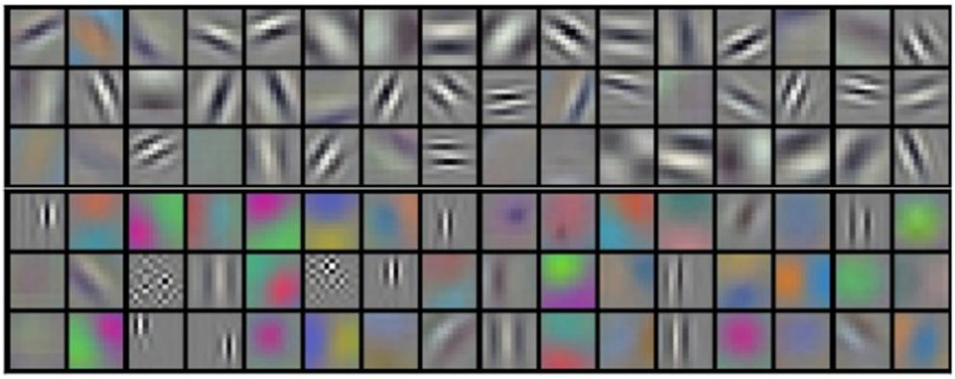
In this case, the input is 1D instead of 2D, and the convolution is strided, i.e., spacing between neuron input patches != 1.

Examples of common filters in image processing

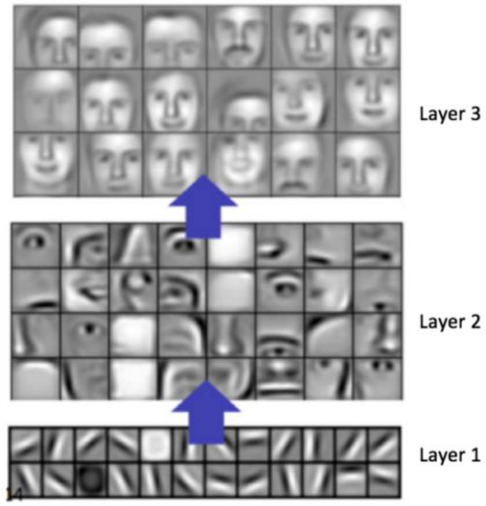
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

Examples of learned filters in image classification



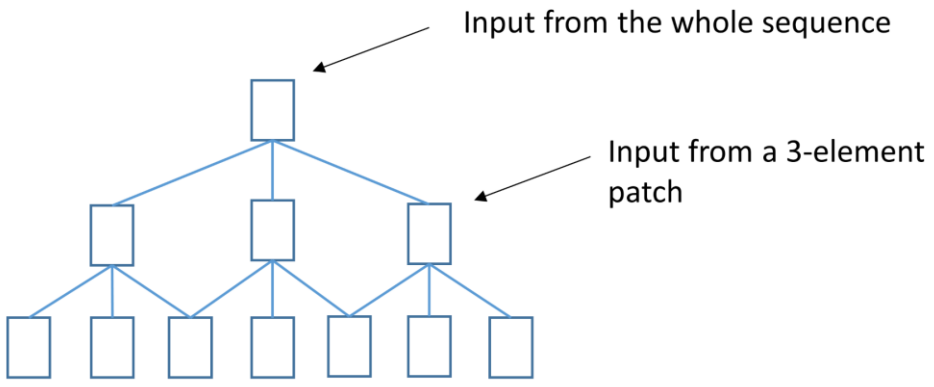
Examples of learned filters in image classification



https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

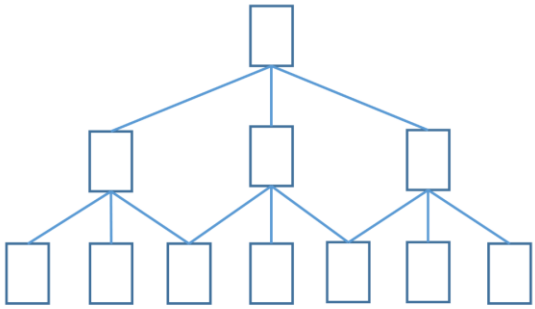
What the filters represent and respond to becomes increasingly complex over the layers

Later layers have larger receptive fields



Why does it work?

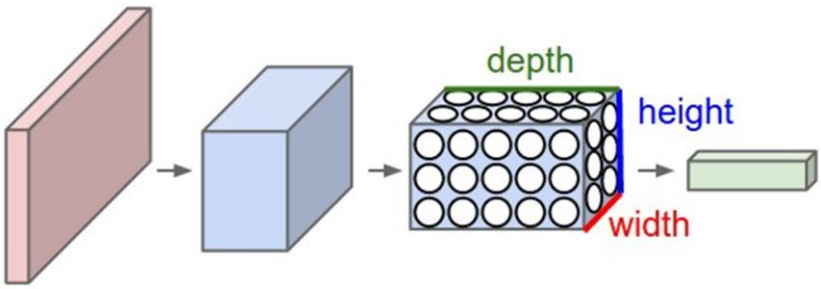
- Weight sharing (the neurons receiving input from different input regions all have the same parameters)
- This greatly reduces the number of optimized parameters in a layer
- At low layers, pixels far from each other are assumed to be more independent



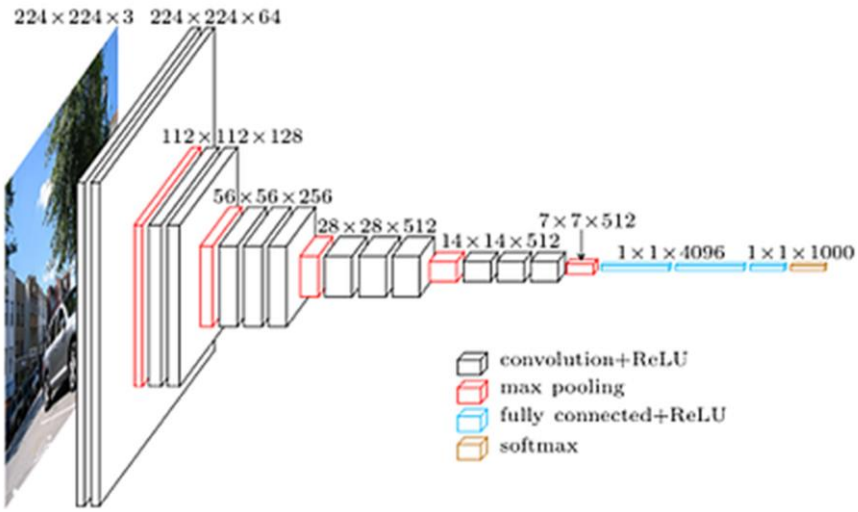
The independence assumption is a good one for natural images, text sequences, and audio. This together with the lower number of parameters makes it more likely that the optimization converges to something reasonable.

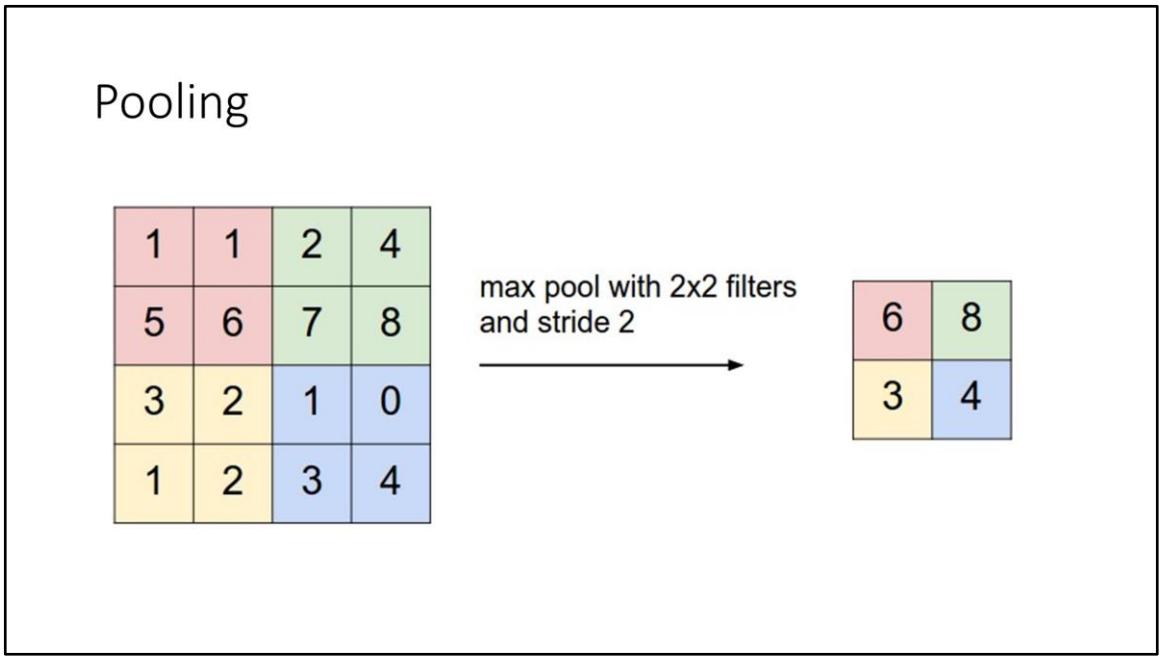
Convolutional networks with images

- Inputs and outputs are 4D tensors with shape (number of images, height,width,depth). Depth=number of image channels
- A single neuron takes as input a multichannel image patch, outputs one pixel in its own channel



The VGG16 model





Pooling denotes various ways of decreasing resolution. Both max-pooling and average-pooling are common.

STRIVING FOR SIMPLICITY: THE ALL CONVOLUTIONAL NET

Jost Tobias Springenberg*, Alexey Dosovitskiy*, Thomas Brox, Martin Riedmiller
Department of Computer Science
University of Freiburg
Freiburg, 79110, Germany
{springj, dosovits, brox, riedmiller}@cs.uni-freiburg.de

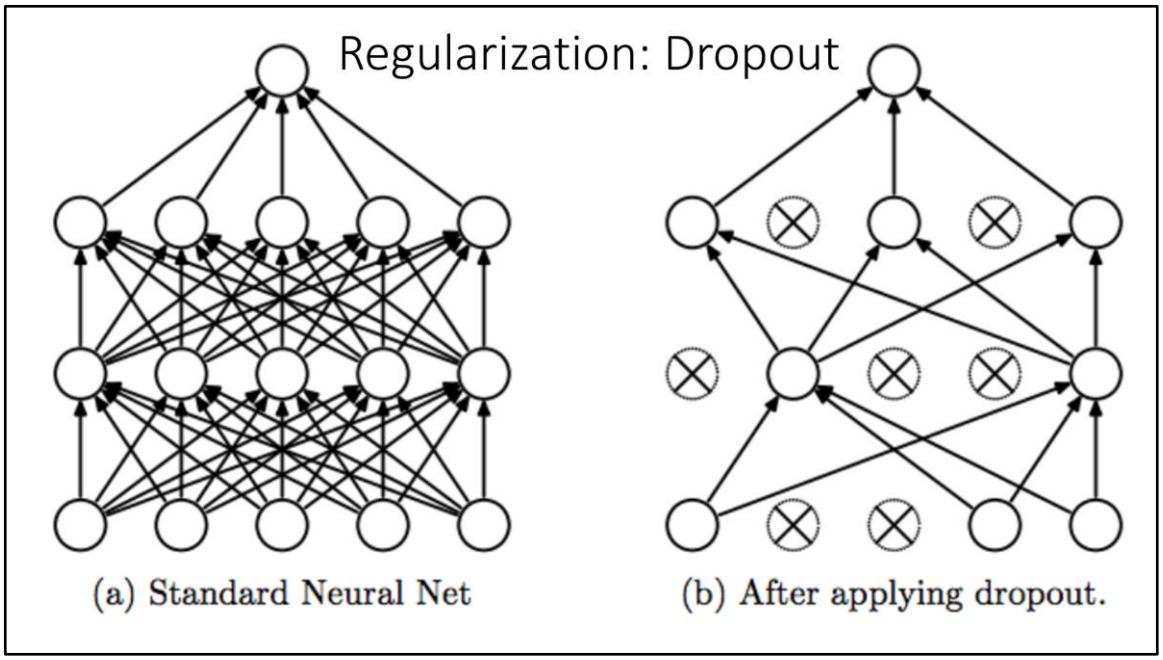
ABSTRACT

Most modern convolutional neural networks (CNNs) used for object recognition are built using the same principles: Alternating convolution and max-pooling layers followed by a small number of fully connected layers. We re-evaluate the state of the art for object recognition from small images with convolutional networks, questioning the necessity of different components in the pipeline. We find that max-pooling can simply be replaced by a convolutional layer with increased stride without loss in accuracy on several image recognition benchmarks. Following this finding – and building on other recent work for finding simple network structures – we propose a new architecture that consists solely of convolutional layers and yields competitive or state of the art performance on several object recognition datasets (CIFAR-10, CIFAR-100, ImageNet). To analyze the network we introduce a new variant of the "deconvolution approach" for visualizing features learned by CNNs, which can be applied to a broader range of network structures than existing approaches.

Although maxpooling has been common since its introduction in the first convolutional neural network to win the ImageNet challenge, later research suggests that pooling may not be necessary, and the same results can be obtained simply by using strided convolution, i.e., sliding the convolution filter over the input array with steps larger than 1. This is also faster.

```
model = keras.models.Sequential()
model.add(Conv2D(16, kernel_size=(5, 5), strides=[2,2],
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(32, (5, 5), activation='relu', strides=[2,2]))
model.add(Conv2D(32, (3, 3), activation='relu', strides=[2,2]))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

With strided convolution, here's all the code one needs to build a convolutional image recognition network with Python and Keras



The code of the previous slide also utilizes dropout layers. These are common in preventing the network from overfitting. In essence, dropout means that for each input image, some randomly selected neurons are “dead” with no activations. This forces the network to learn robust representations that do not depend on any single neuron.

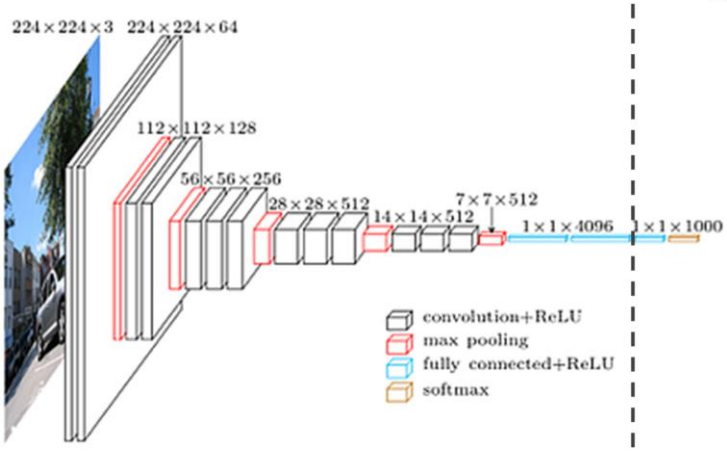
Srivastava, Nitish, et al. ”Dropout: a simple way to prevent neural networks from overfitting”, JMLR 2014

Contents

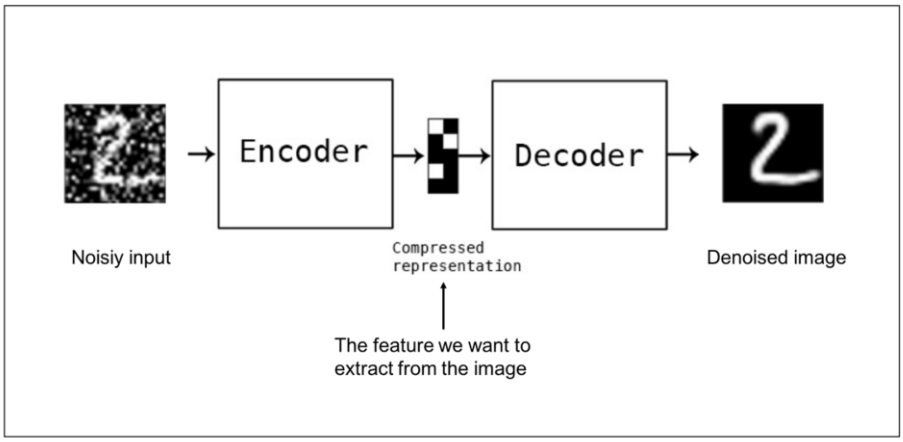
- Preliminaries: compute graphs, artificial neurons, activation functions, loss functions
- Understanding nonlinear activations
- Understanding skip-connections
- Convolutional neural networks
- **Encoder-decoder architectures**

Encoder, a compact representation of image contents

Simple classifier: One neuron per Image class



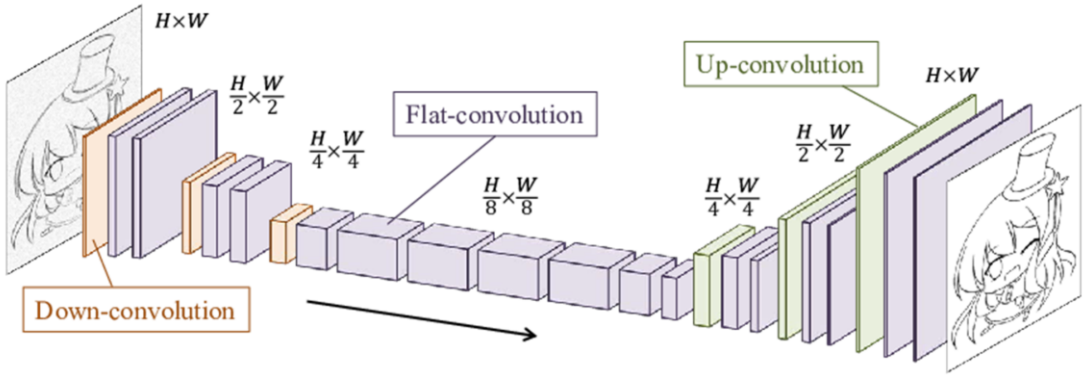
Encoder-decoder architectures



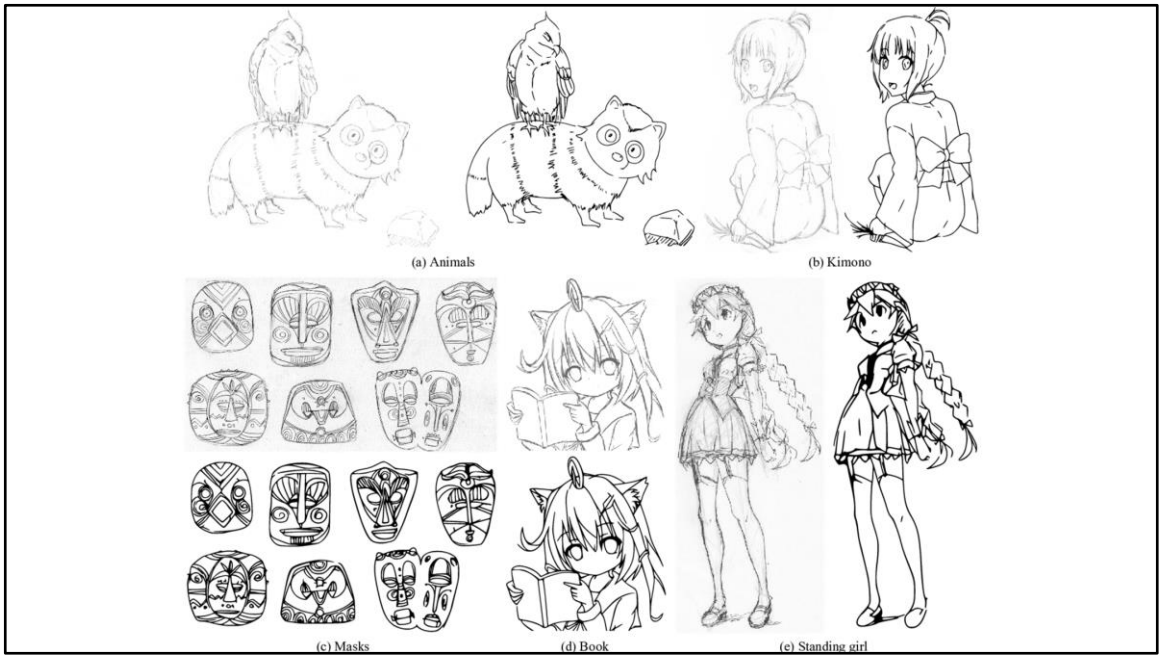
https://blog.sicara.com/keras-tutorial-content-based-image-retrieval-convolutional-denoising-autoencoder-dc91450cc511

The decoder network learns to invert what the encoder does, i.e., convert the internal, abstract representation back to the input space, in this case the images. This kind of an encoder-decoder network is typically trained by feeding the same images as both inputs and targets, possibly corrupting one or the other with noise. Since the compressed representation learns to focus on the essential features, this produces a denoising effect.

Convolutional encoder-decoder

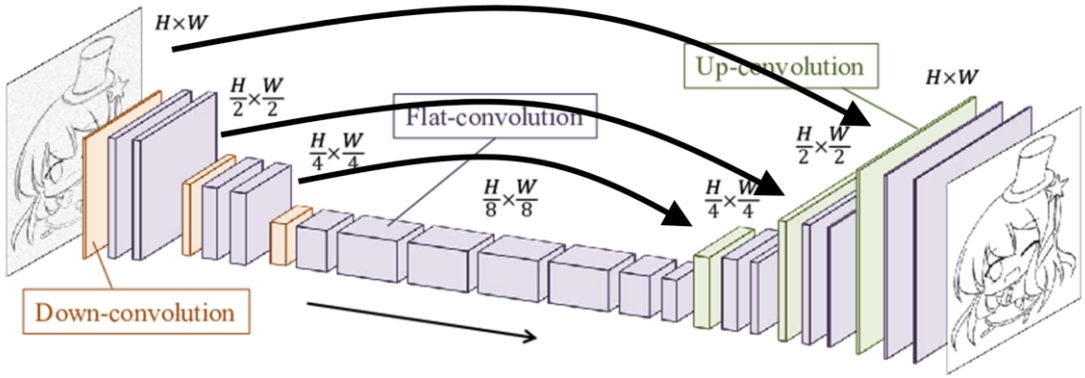


Trained with pencil-ink pairs,
http://hi.cs.waseda.ac.jp/%7Eesimo/en/research/sketch/



results

Skip-connections and convolutions:
high-resolution local decisions utilizing local input info



Skip-connections and convolutions: U-Net (2015)

U-Net: Convolutional Networks for Biomedical Image Segmentation

Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and BIOSS Centre for Biological Signalling Studies,
University of Freiburg, Germany
ronneber@informatik.uni-freiburg.de,
WWW home page: http://lmb.informatik.uni-freiburg.de/

Abstract. There is large consent that successful training of deep networks requires many thousand annotated training samples. In this paper, we present a network and training strategy that relies on the strong use of data augmentation to use the available annotated samples more efficiently. The architecture consists of a contracting path to capture context and a symmetric expanding path that enables precise localization. We show that such a network can be trained end-to-end from very few images and outperforms the prior best method (a sliding-window convolutional network) on the ISBI challenge for segmentation of neuronal structures in electron microscopic stacks. Using the same network trained on transmitted light microscopy images (phase contrast and DIC) we won the ISBI cell tracking challenge 2015 in these categories by a large margin. Moreover, the network is fast. Segmentation of a 512x512 image takes less than a second on a recent GPU. The full implementation (based on Caffe) and the trained networks are available at http://lmb.informatik.uni-freiburg.de/people/ronneber/u-net.

This is the original paper that proposed a convolutional encoder-decoder network with skip.connections

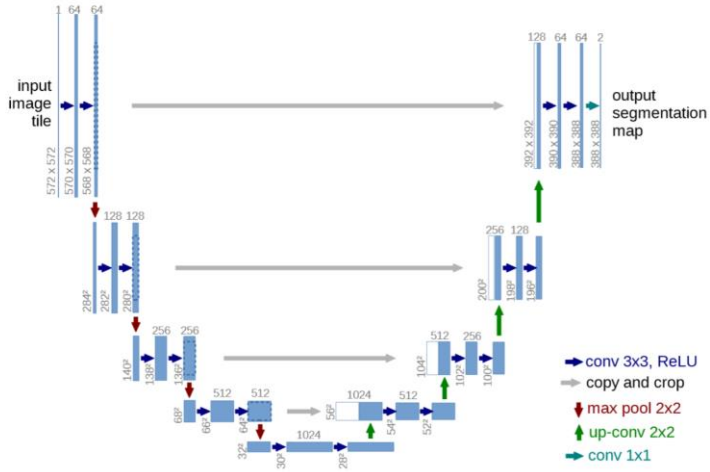


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

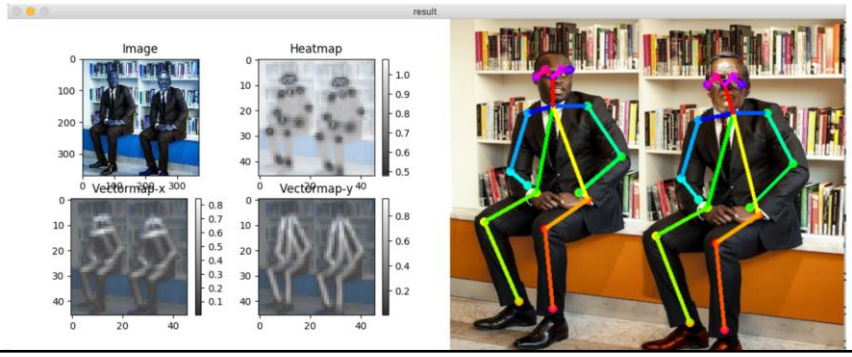
Pixels to pixels is easy

- For a convolutional neural network, it may be easier to map input pixels to output pixels instead of arbitrary values.
- Especially true for U-Nets
- For example, easier to color a recognized/tracked object with some predetermined color than to output the object's coordinates
- Better to encode everything as pixels than mix data types
- Local decisions based local information are easier to learn

For example, a Reinforcement Learning agent may learn better from pixels with a convolutional neural network if it can make decisions based on local data, e.g., an enemy getting close (the alternative being to pack all game state into a vector).

OpenPose

- Real-time tracking of multiple people, works even with a moving camera
- First step: pixel-to-pixel labeling of body parts
- https://github.com/ildoonet/tf-pose-estimation
- Test this in your browser (also works on mobile): https://urly.fi/1dfJ



Now that Kinect is no longer available, this is the best alternative.

ml5js browser-based demos and tutorial:

https://ml5js.org/docs/PoseNet
https://editor.p5js.org/genekogan/sketches/Hk2Q4Sqe4
https://editor.p5js.org/AndreasRef/sketches/r1_w73FhQ

Summary

- Preliminaries: compute graphs, artificial neurons, activation functions, loss functions
- Understanding nonlinear activations
- Understanding skip-connections
- Convolutional neural networks
- Encoder-decoder architectures

We're now at the end of this lecture, time to summarize some key takeaways

Key takeaways

- Preliminaries: **Neural networks are compute graphs.** Tensorflow, Pytorch etc. are tools for building and optimizing the graphs
- Loss functions: **Mean squared error for regression** (line fitting as the simple example)**, softmax cross-entropy for classification** (and other tasks where last layer outputs a discrete probability distribution)
- Understanding nonlinear activations: **Deeper networks can partition the input space in exponentially more detailed manner**, but width also helps and makes optimizing network parameters easier.
- Understanding skip-connections: **Key to training very deep networks, makes the optimization landscape more smooth**
- Convolutional neural networks: **Useful whenever input variables close to each other (in space or time) have stronger relationships**
- Encoder-decoder architectures: **The U-Net** is the standard tool in mapping pixels to pixels (or audio to audio) in a deterministic way, i.e., with clear ground truth output. If there are multiple possible outputs, one needs a generative model like GANs (next lecture)