Artificial Neural Networks

V. Lyubchich

2020-04-27

(Some of the) Resources

- Playlist of coding neural networks in Python from scratch https://www.youtube.com/playlist?list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3
- Chapter 11 of Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media.
- LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436-444. doi: 10.1038/nature14539
- + other references from each slide

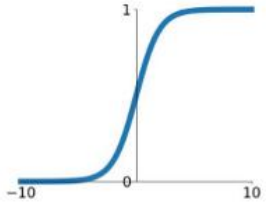
Outline

1. “Forward pass” (feedforward neural network)
   - Inputs, outputs, and hidden layers
   - Activation functions
2. Back-propagation (training)
   - Dropout
3. Overview of extensions

Activation functions – some popular examples (need to be non-linear and simple to compute; allow to compute the derivative)

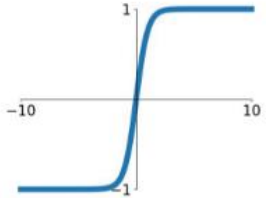
Logistic = **Sigmoid**

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



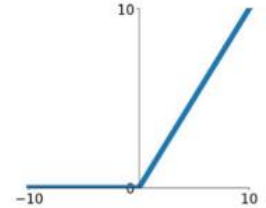
tanh

$$tanh(x) = \frac{2}{1+e^{-2x}} - 1$$



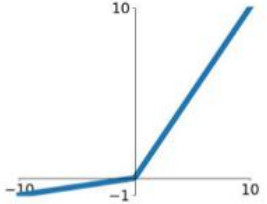
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



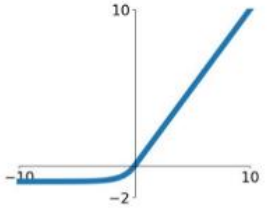
special case

Maxout ***learnable activation function**

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

ELU

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



ReLU = Rectified Linear Unit; ELU = Exponential Linear Unit

https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

Variants of sigmoid

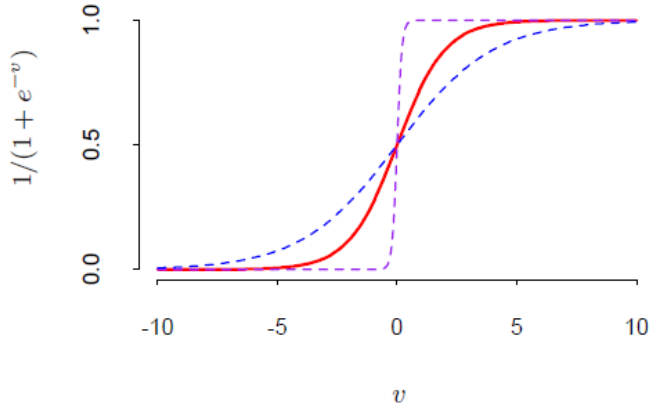


FIGURE 11.3. *Plot of the sigmoid function* $\sigma(v) = 1/(1+\exp(-v))$ *(red curve), commonly used in the hidden layer of a neural network. Included are* $\sigma(sv)$ *for* $s = \frac{1}{2}$ *(blue curve) and* $s = 10$ *(purple curve). The scale parameter* s *controls the activation rate, and we can see that large* s *amounts to a hard activation at* $v = 0$*. Note that* $\sigma(s(v - v_0))$ *shifts the activation threshold from 0 to* v_0*.*

Hastie et al. (2009)

Softmax function
(usually used in the output layer
for multiclass classification)

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for $i = 1, \ldots, K$ and $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$

K is the number of classes.

Can be also

$$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}} \text{ or } \sigma(\mathbf{z})_i = \frac{e^{-\beta z_i}}{\sum_{j=1}^{K} e^{-\beta z_j}}$$

```
>>> import numpy as np
>>> a = [1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0]
>>> np.exp(a) / np.sum(np.exp(a))
array([0.02364054, 0.06426166, 0.1746813, 0.474833, 0.02364054,
       0.06426166, 0.1746813])
```

https://en.wikipedia.org/wiki/Softmax_function

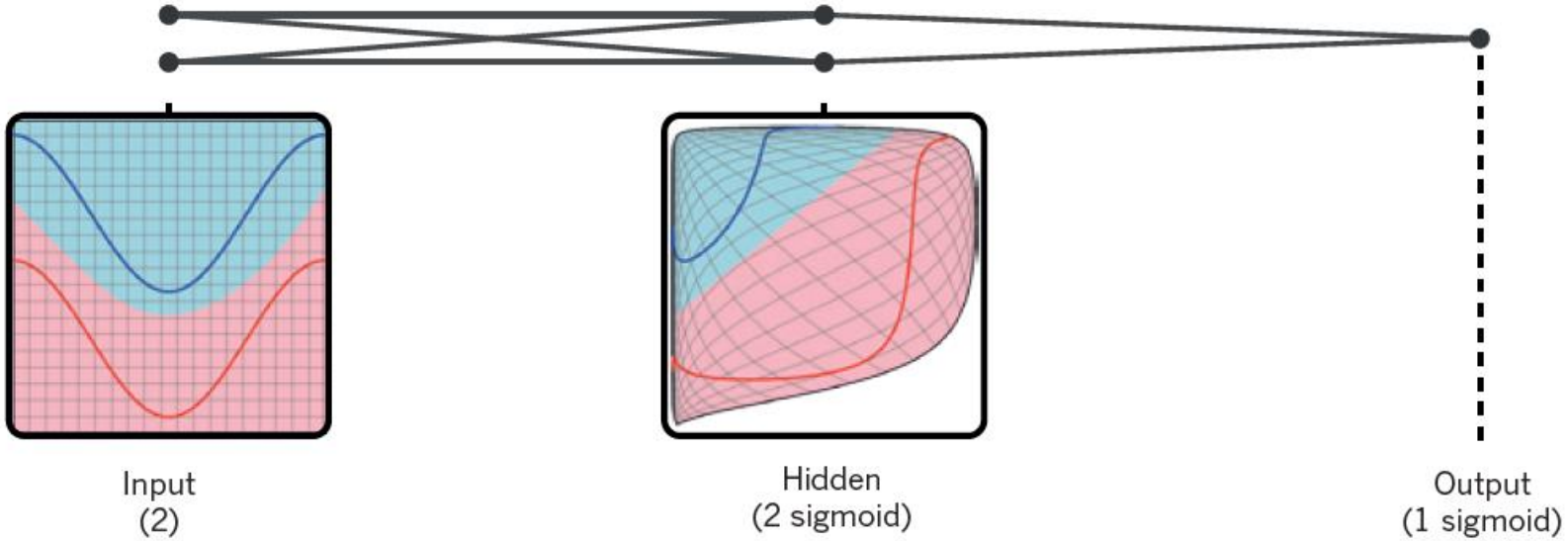
One-hot encoding

CompanyName	Categoricalvalue	Price
VW	1	20
Acura	2	10011
Honda	3	50000
Honda	3	10000

VW	Acura	Honda	Price
1	0	0	20000
0	1	0	10011
0	0	1	50000
0	0	1	10000

https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

The effect of transformations



A multi-layer neural network (shown by the connected dots) can distort the input space to make the classes of data (examples of which are on the red and blue lines) linearly separable. Note how a regular grid (shown on the left) in input space is also transformed (shown in the middle panel) by hidden units. This is an illustrative example with only two input units, two hidden units and one output unit, but the networks used for object recognition or natural language processing contain tens or hundreds of thousands of units. Reproduced with permission from C. Olah (http://colah.github.io/).

LeCun et al. (2015)

Best practices of using activation functions

- Use **ReLU** in hidden layer activation, but be careful with the learning rate and monitor the fraction of dead units.
- If **ReLU** is giving problems. Try Leaky ReLU, PReLU, Maxout. Do not use sigmoid.
- **Normalize the data to achieve higher validation accuracy, and standardize if you need the results faster.**
- The **sigmoid** and **hyperbolic tangent** activation functions cannot be used in networks with many layers due to the vanishing gradient problem.

https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044

2. Back-propagation (training)

- In the forward pass, the inputs vary, but the weights are fixed
- Then, compute the errors (loss function): For classification, this is usually cross entropy (XC, log loss), while for regression it is usually squared error loss (SEL)
- In back-propagation, inputs-outputs are fixed but weights vary (to compute gradient/derivative of the loss); see e.g., https://www.youtube.com/watch?v=GlcnxUlrtek
- Subtract the scalar * gradient from the weights to move 'downhill' (gradient descent method). scalar is called the learning rate. The learning rate can be adaptive.

Part of the "backward pass"

Given these derivatives, a gradient descent update at the $(r+1)$st iteration has the form

Updated weights

Derivative of error with respect to the weights

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}},$$

$$\alpha_{m\ell}^{(r+1)} = \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}}, \quad (11.13)$$

where γ_r is the *learning rate*

The two-pass procedure is what is known as back-propagation.
It has also been called the *delta rule*.

The updates in (11.13) are a kind of *batch learning*, with the parameter updates being a sum over all of the training cases. Batch size becomes another tuning parameter.

Hastie et al. (2009)

Dropout – similar to regularization in regression

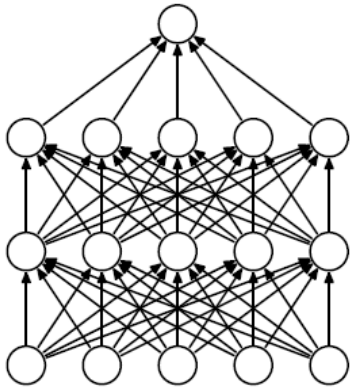
Consider a neural network with L hidden layers. Let $l \in \{1, \ldots, L\}$ index the hidden layers of the network. Let $\mathbf{z}^{(l)}$ denote the vector of inputs into layer l, $\mathbf{y}^{(l)}$ denote the vector of outputs from layer l ($\mathbf{y}^{(0)} = \mathbf{x}$ is the input). $W^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases at layer l. The feed-forward operation of a standard neural network (Figure 3a) can be described as (for $l \in \{0, \ldots, L-1\}$ and any hidden unit i)

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

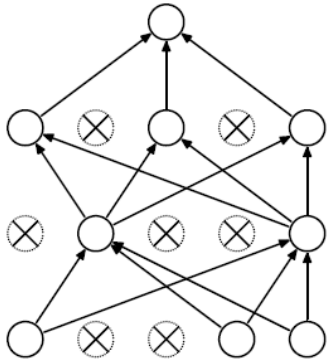
where f is any activation function, for example, $f(x) = 1/(1 + \exp(-x))$.

With dropout, the feed-forward operation becomes (Figure 3b)

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$



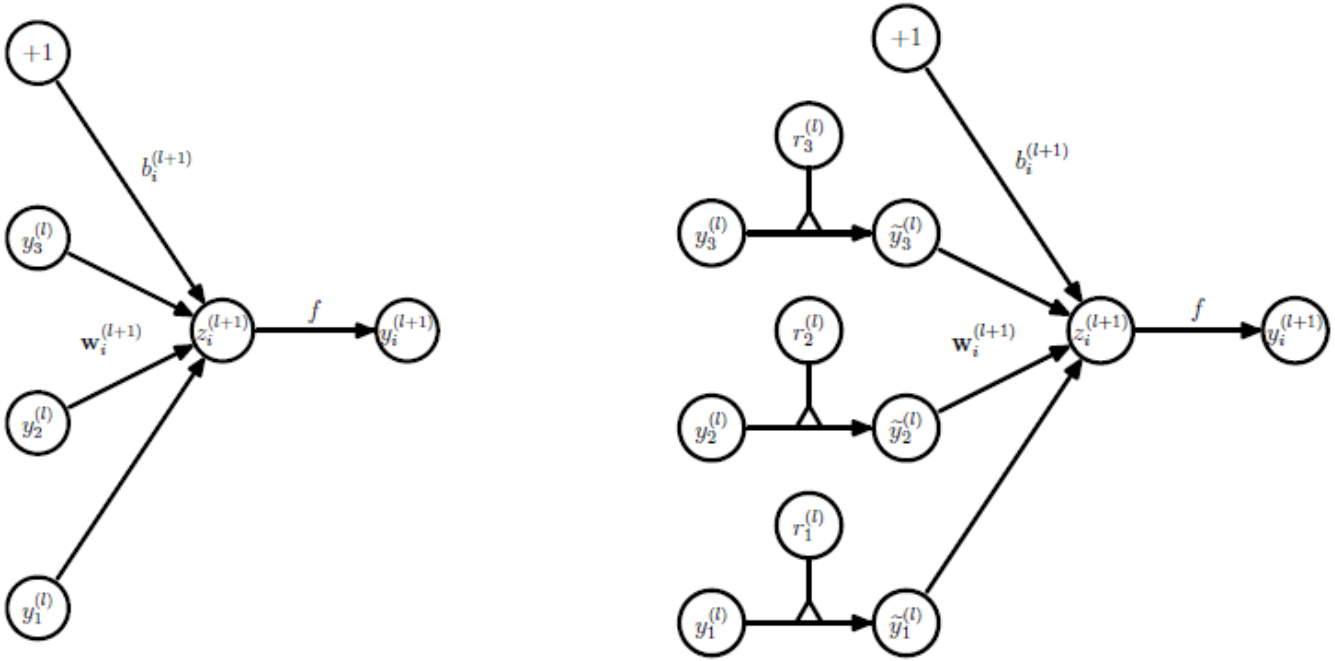
(a) Standard Neural Net



(b) After applying dropout.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

Dropout (continued)



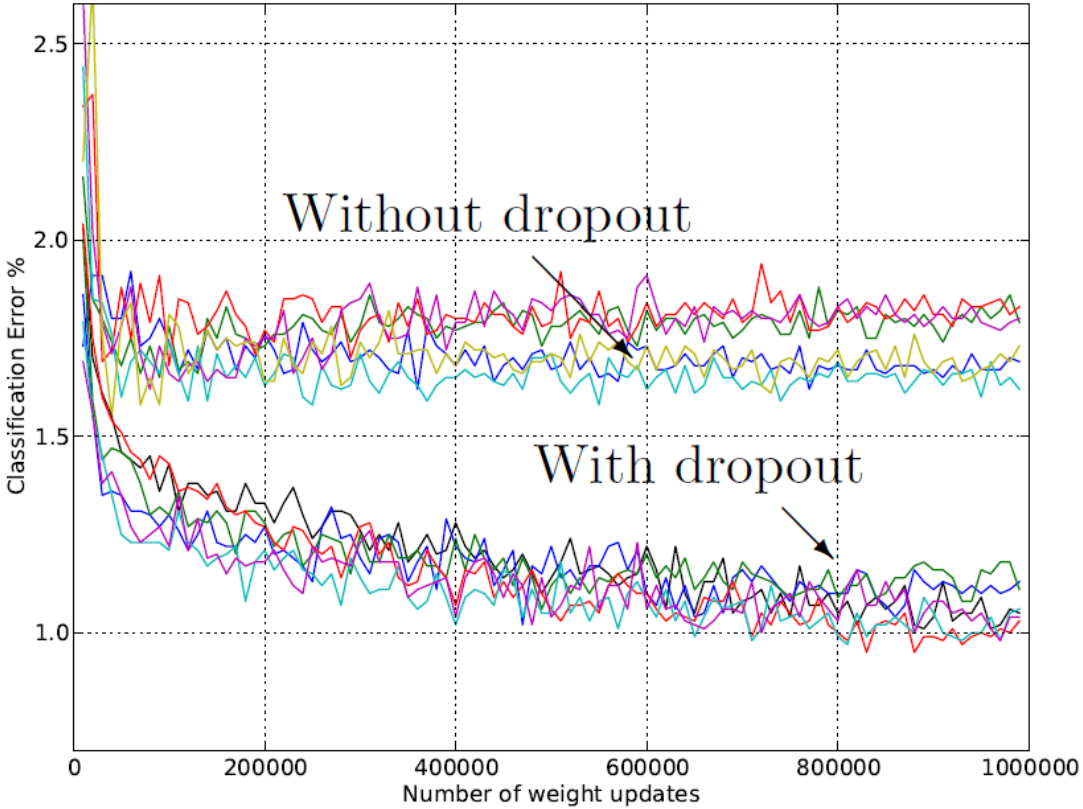
(a) Standard network

(b) Dropout network

Figure 3: Comparison of the basic operations of a standard and dropout network.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

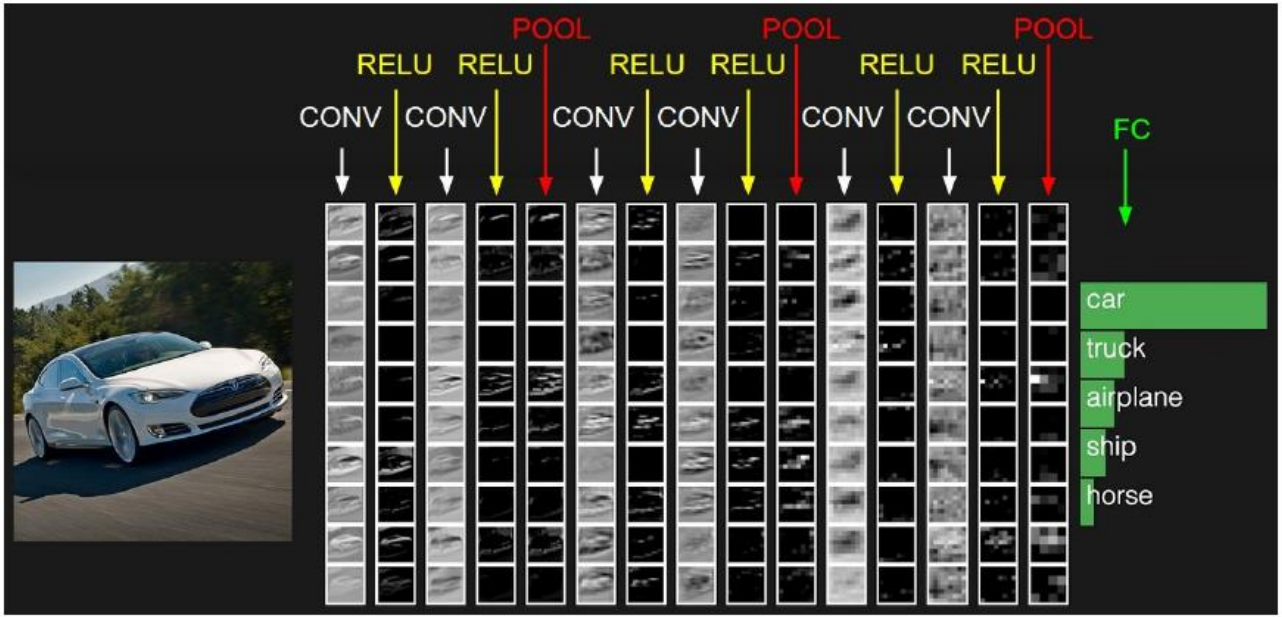
Dropout (continued)



Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

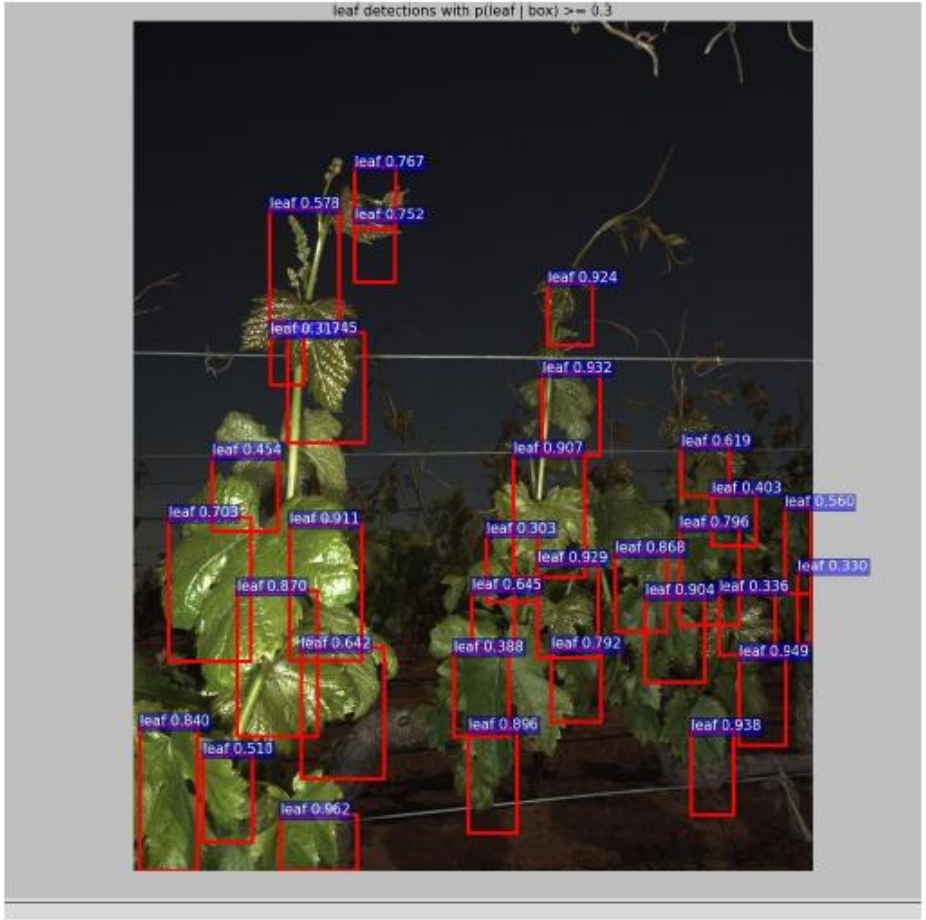
3. Overview of extensions

- **Examples of problems: Classification**



This and following examples are taken from lectures by Dr. Alex Ter-Sarkisov, City U.of London

- **Examples of problems: Object Detection**

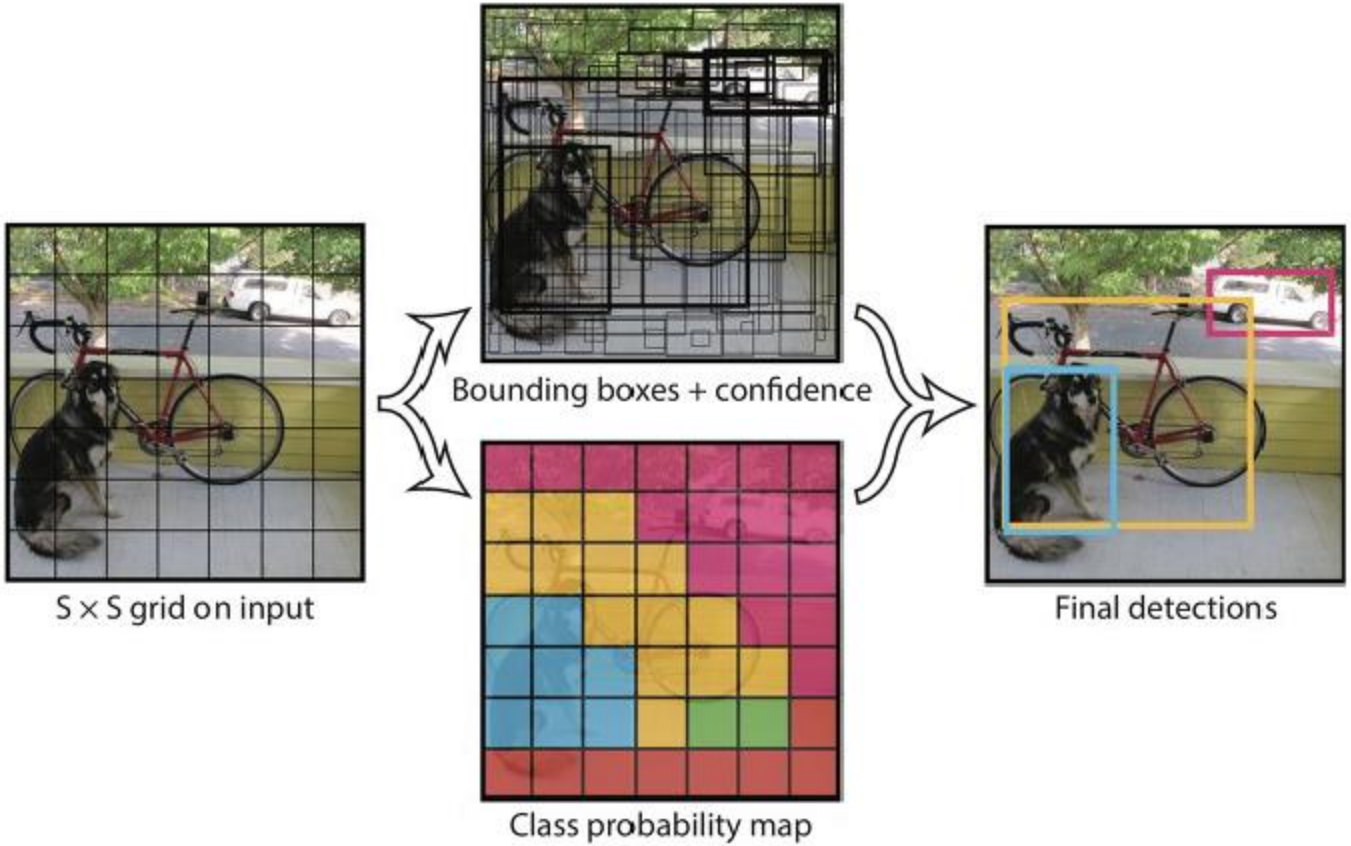


- **Examples of problems: Semantic Segmentation**

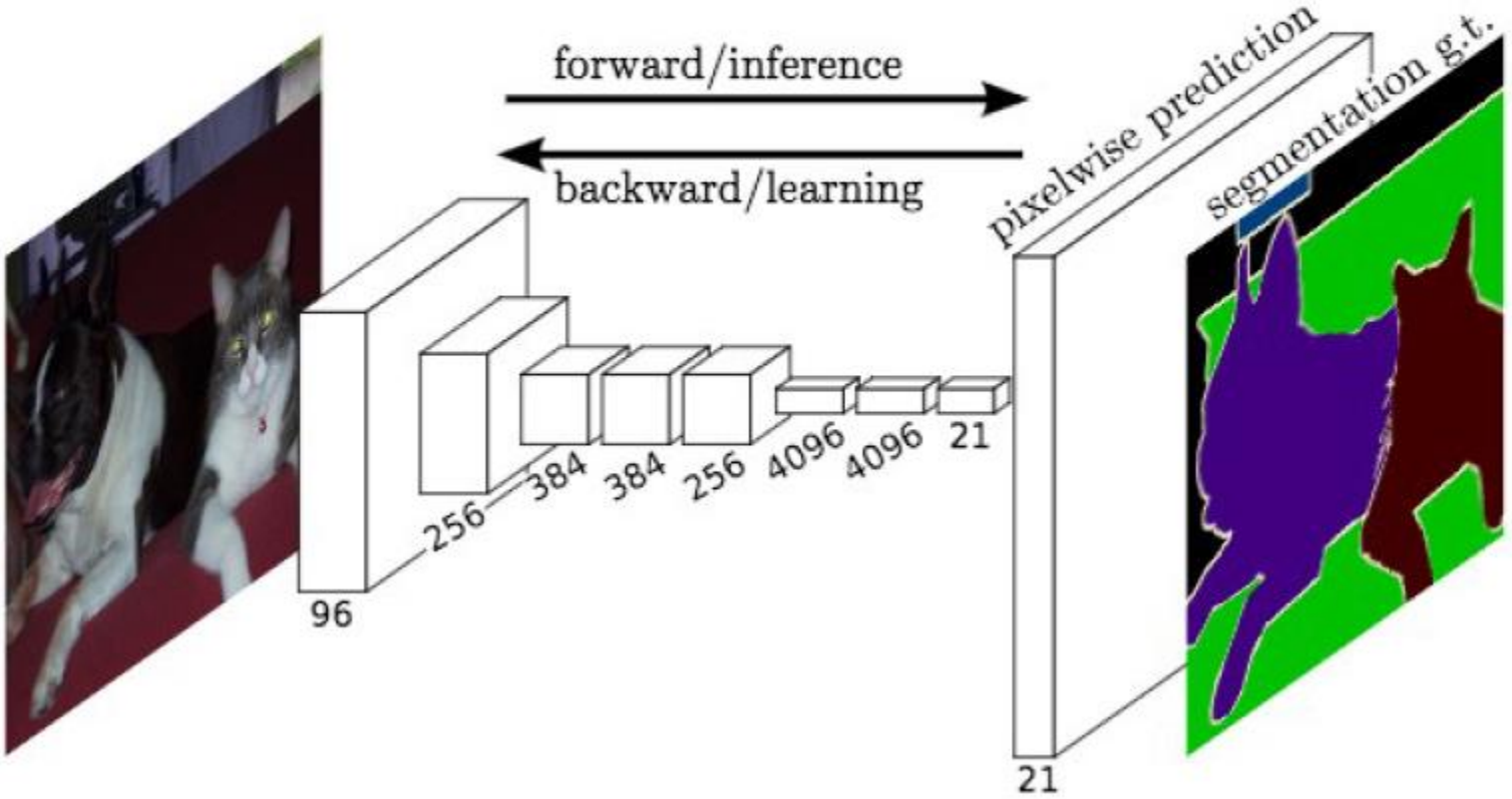
- **Examples of problems: Instance Segmentation**

- **Examples of problems: Style Transfer**

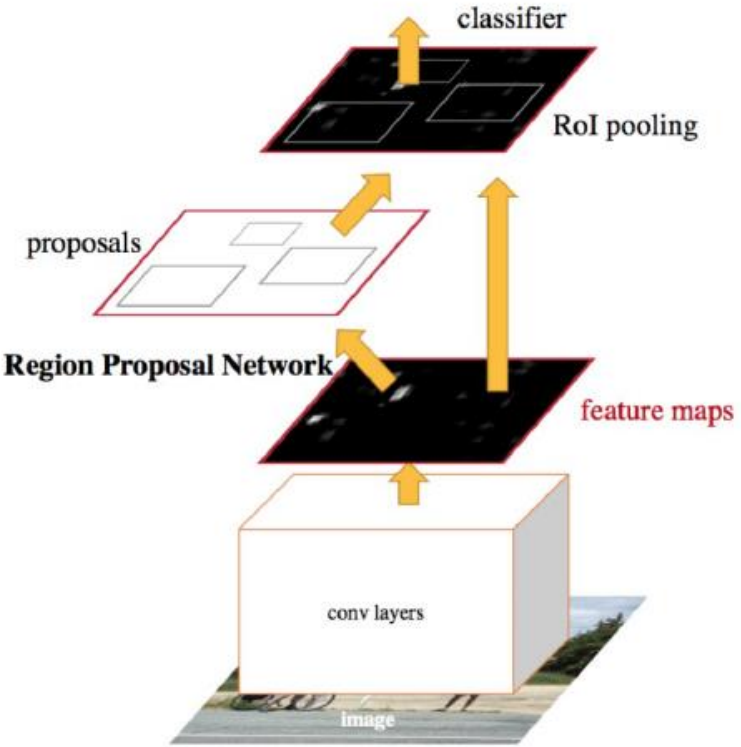
- **Examples of models: YOLO (object detection)**

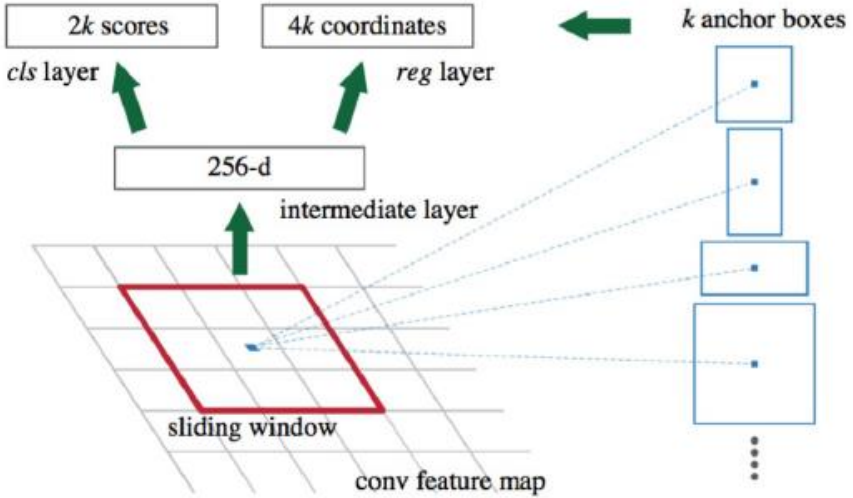


- **Examples of models: FCN (semantic segmentation)**

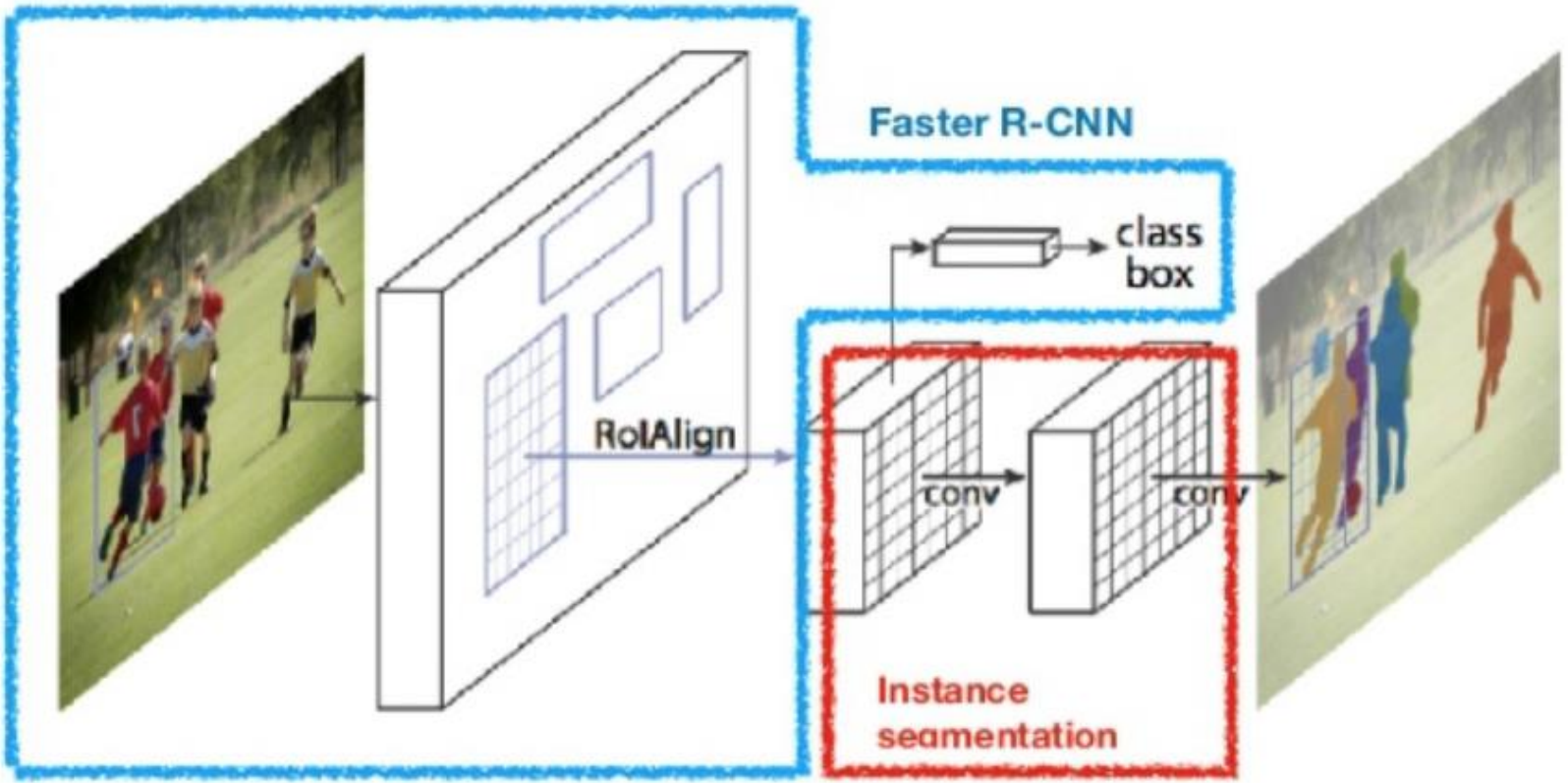


- **Examples of models: RCNN (object detection)**



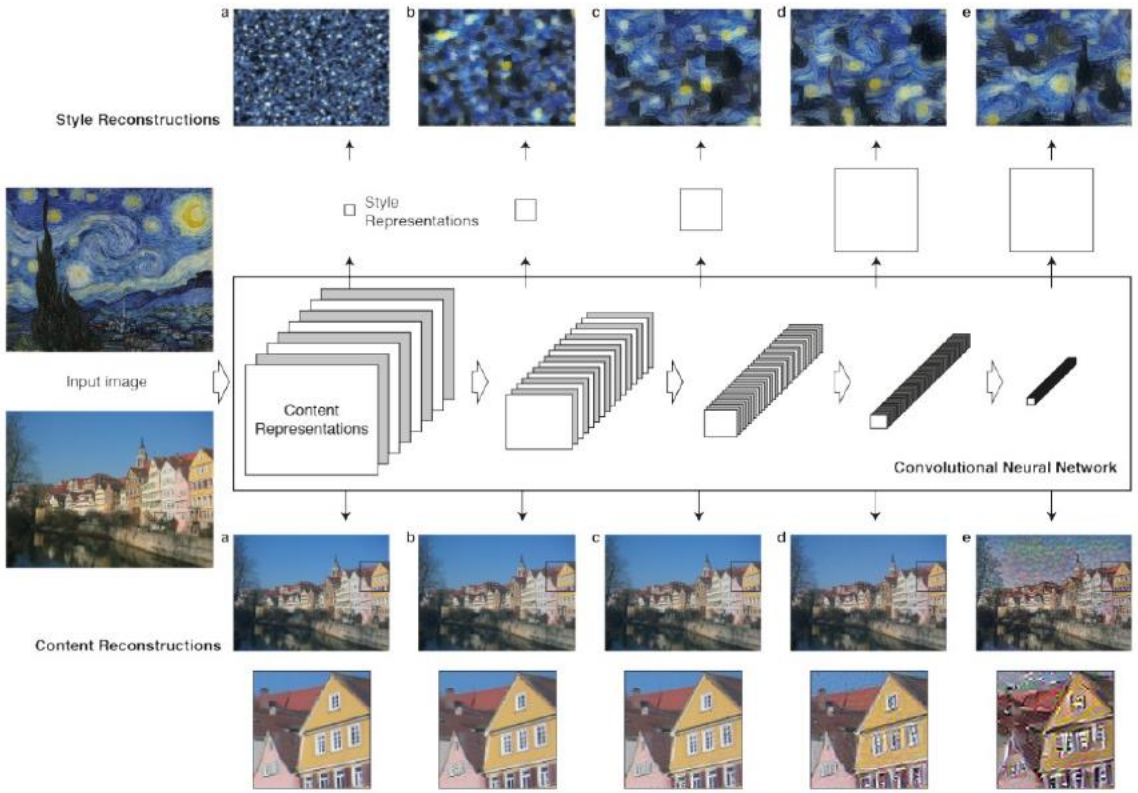


- **Examples of models: Mask R-CNN (instance segmentation)**

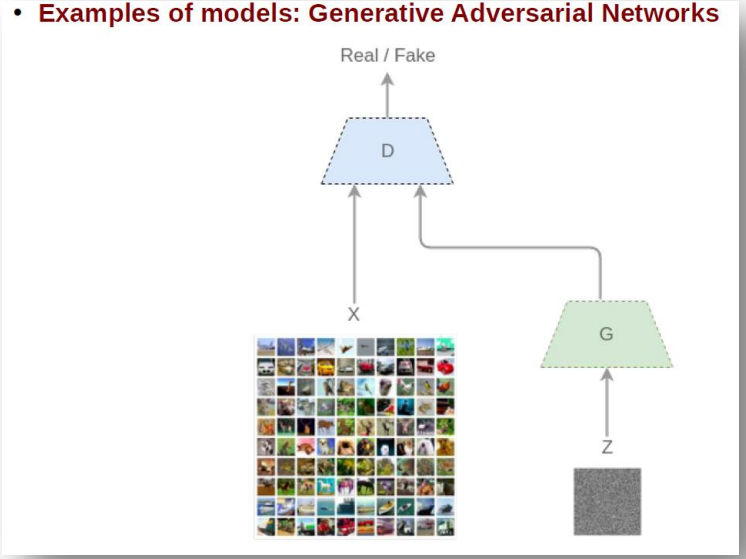


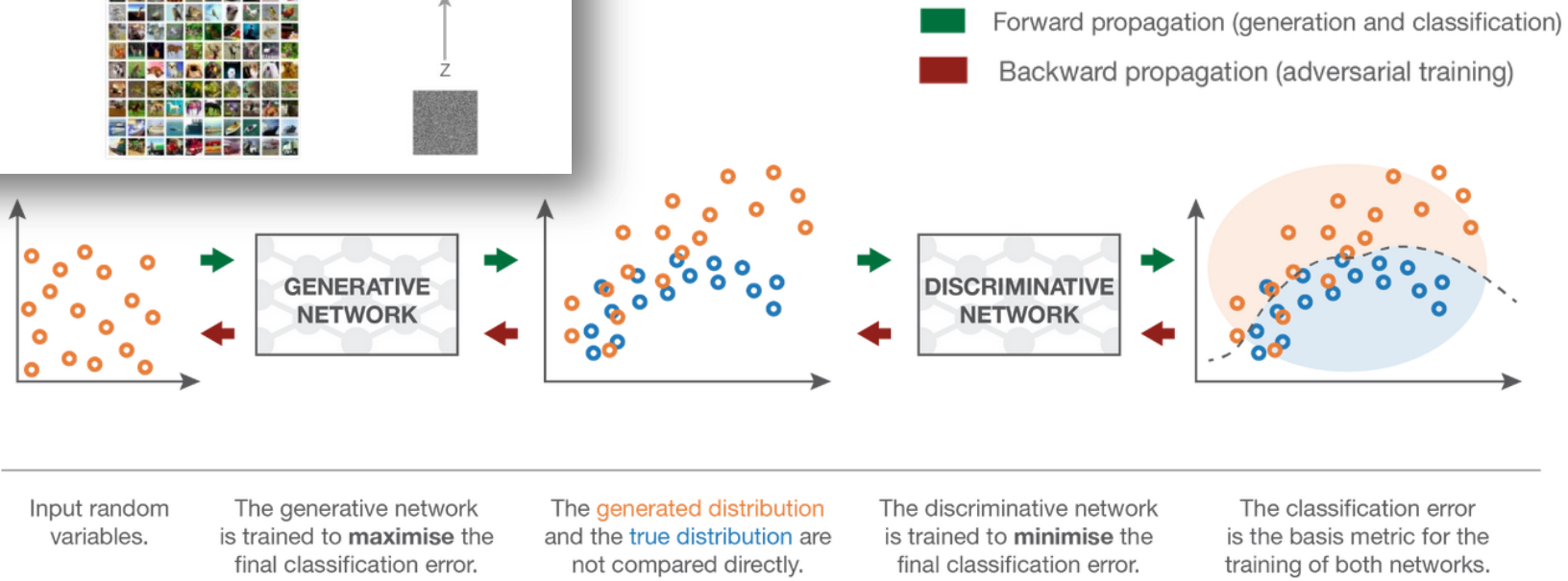
RoI = Region of Interest; R-CNN = regional convolutional neural network (selective search)

- **Examples of models: Generative Networks**



- **Examples of models: Generative Adversarial Networks**





https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29

Coding

- Pytorch, a novel framework for training, deploying and scoring models on GPUs
- Tensorflow, keras – can be run from R too (e.g., see https://www.r-bloggers.com/convolutional-neural-networks-in-r/ https://letyourmoneygrow.com/2018/05/27/classifying-time-series-with-keras-in-r-a-step-by-step-example/)
- Python with a range of modules (numpy, scipy, opencv, matplotlib, scikit-learn, etc.)
- Some auxiliary tools, e.g., interfaces for loading datasets
- H2o.ai – R package available (https://www.h2o.ai)

Coding (continued)

Pytorch

- State-of-the-art for DL modeling (v.1.1.0),
- Uses tensors to backpropagate,
- Integrated with Python: easy transition between Pytorch tensors and NumPy arrays,
- Open source: inherit Pytorch modules and implement your own functionality,
- Two mandatory methods for a custom class: __*init*__ and *forward,*
- Built-in basic layers for ConvNets and RNNs,
- Convenient interface for dataset creation and handling, *Dataset* and *DataLoader,*
- *Torchvision* library with easily accessible, built-in models like VGG, ResNet, with pre-trained weights and datasets
- Advanced models for object detection and instance segmentation, like *torchfcn,*
- Easy switching between GPU and CPU interfaces