



From Linear Regression to Deep Neural Networks to Segmentation

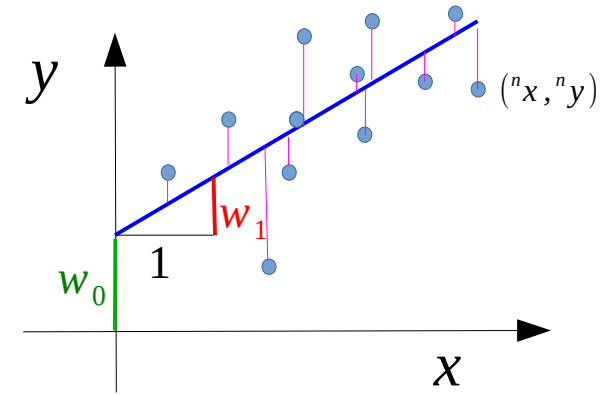
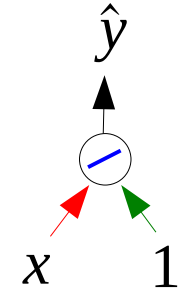
Lucas C Parra

Linear Regression

Data $(^n x, ^n y), n=1 \dots N$

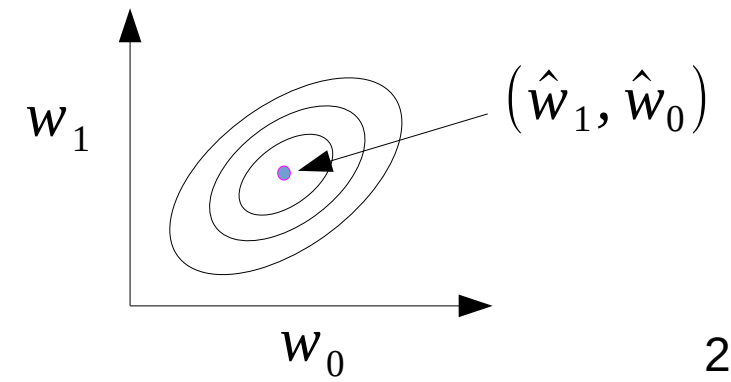
Model $\hat{y} = w_1 x + w_0$

Error $e = \hat{y} - y$



Cost function Mean square error $E(w_1, w_0) = \sum_n (^n e)^2$

Learning $\hat{w}_1, \hat{w}_0 = \underset{w_1, w_0}{\operatorname{argmin}} E(w_1, w_0)$



Revisiting segmentation by fitting the background intensity and thresholding:

Linear model with quadratic error

$$E(\mathbf{w}) = \sum_n ({}^n y - \mathbf{w}^T \cdot {}^n \mathbf{x})^2$$

has closed form solution:

$$\hat{\mathbf{w}} = \left(\sum_n {}^n \mathbf{x} \cdot {}^n \mathbf{x}^T \right)^{-1} \cdot \left(\sum_n {}^n \mathbf{x} \cdot {}^n y \right)$$

$$= R_{xx}^{-1} \cdot R_{xy} = \mathbf{x} \setminus \mathbf{y}$$

Notice that subtraction and thresholding is a linear operation followed by nonlinearity

```
>> [X,Y] = meshgrid(1:size(img,2),1:size(img,1));
>> img = img - mycurvefit(img,X,Y); % linear prediction from X,Y
>> img = img > graythresh(img); % nonlinearity
```

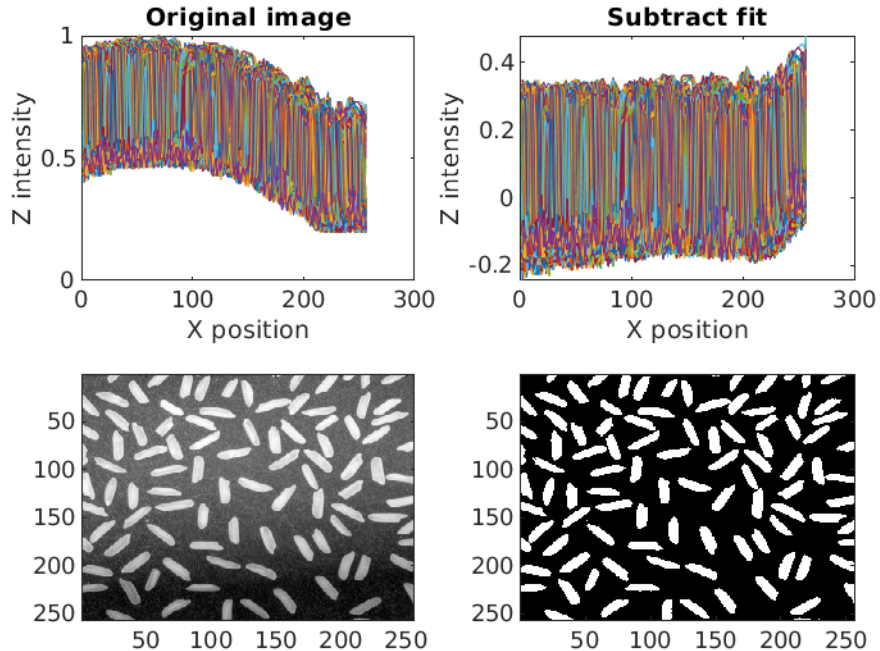
Making position of the input also a feature to predict the output is called “**position encoding**” in machine learning.

```
function Zest = mycurvefit(Z,X,Y)
% Zest = mycurvefit(Z) fit Z to be approximated by
% Zest=w1+w2*X+w3*Y+w4*Y^2 in a least squares sense.
dims = size(Z); [X,Y] = meshgrid(1:dims(2),1:dims(1));
woptimal = [ones(size(Z(:))) X(:) Y(:) Y(:).^2]\Z(:);
Zest = mycurve(woptimal,X,Y);
```

```
function Zest = mycurve(p,X,Y);
Zest = w(1)+w(2)*X+w(3)*Y+w(4)*Y.^2;
```

This segmentation code could be called a “single layer dense network with position encoding”.

Revisiting segmentation by fitting the background intensity and thresholding:



Notice that subtraction and thresholding is a linear operation followed by nonlinearity

```
>> [X,Y] = meshgrid(1:size(img,2),1:size(img,1));
>> img = img - mycurvefit(img,X,Y); % linear prediction from X,Y
>> img = img > graythresh(img); % nonlinearity
```

Making position of the input also a feature to predict the output is called “**position encoding**” in machine learning.

```
function Zest = mycurvefit(Z,X,Y)
% Zest = mycurvefit(Z) fit Z to be approximated by
% Zest=w1+w2*X+w3*Y+w4*Y^2 in a least squares sense.
dims = size(Z); [X,Y] = meshgrid(1:dims(2),1:dims(1));
woptimal = [ones(size(Z(:))) X(:) Y(:) Y(:).^2]\Z(:);
Zest = mycurve(woptimal,X,Y);
```

```
function Zest = mycurve(p,X,Y);
Zest = w(1)+w(2)*X+w(3)*Y+w(4)*Y.^2;
```

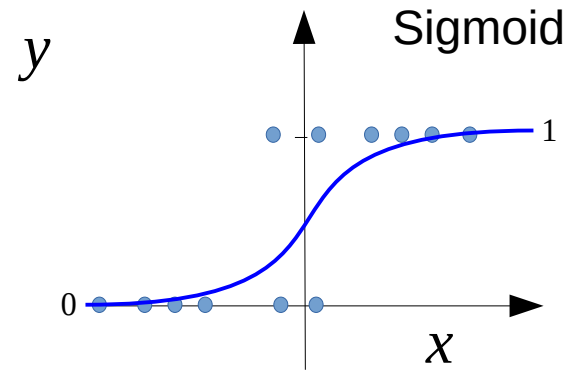
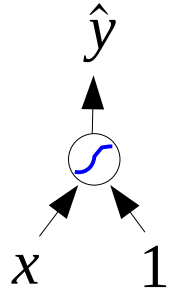
This segmentation code could be called a “single layer dense network with position encoding”.

Logistic Regression

Data $(^n x, ^n y), n=1 \dots N$

Model $\hat{y} = f(w_1 x + w_0)$

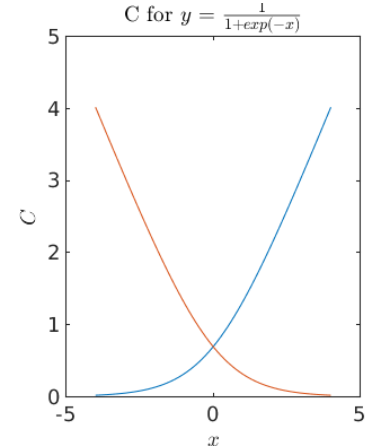
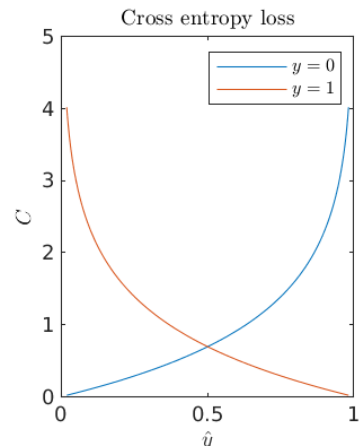
Sigmoid $\hat{y} = f(x) = \frac{\exp(x)}{1 + \exp(x)}$



Cost (binary cross entropy) $c = -y \log \hat{y} - (1 - y) \log (1 - \hat{y})$

Cost function $E(w_1, w_0) = \sum_n c$

Learning $\hat{w}_1, \hat{w}_0 = \underset{w_1, w_0}{\operatorname{argmin}} E(w_1, w_0)$



Perceptron

Data $(^n x_1 \dots ^n x_{D_x}, ^n y_1 \dots ^n y_{D_y}), n=1 \dots N$

Model $\hat{y}_i = f_i(\sum_j w_{ij} x_j)$

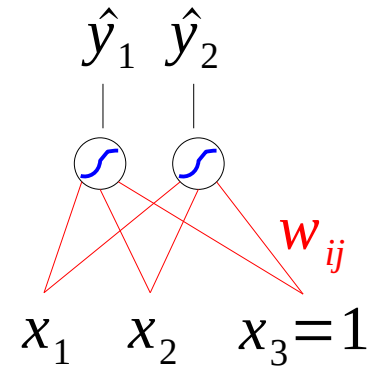
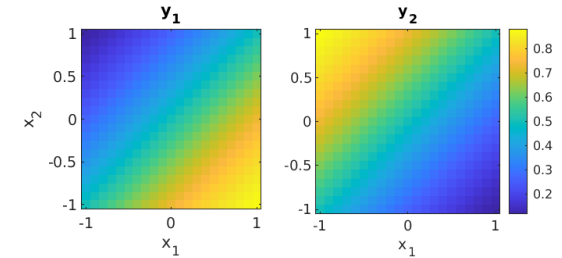
Softmax $\hat{y}_i = f_i(x_1 \dots x_D) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$ ← Think of x_i as the strength of evidence for class i

Cost $c_i = -y_i \log \hat{y}_i$
(categorical cross entropy)

Cost function $E(w_{ij}) = \sum_n \sum_i c_i$

Learning $\hat{w}_{ij} = \underset{w_{ij}}{\operatorname{argmin}} E(w_{ij})$

Softmax



“Dense network layer”



Gradient Descent

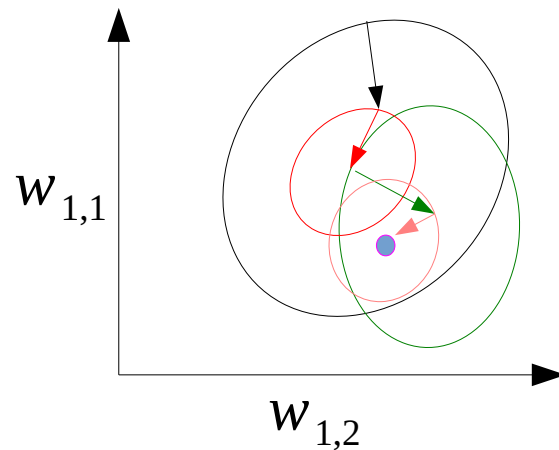
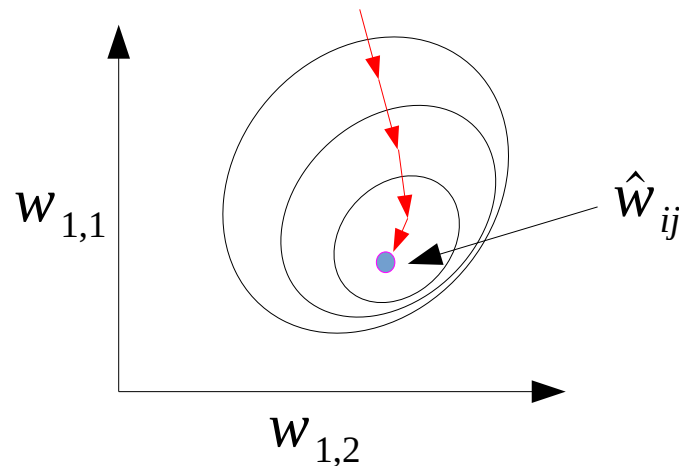
$$\hat{w}_{ij} = \underset{w_{ij}}{\operatorname{argmin}} E(w_{ij})$$

Gradient descent

$$\Delta w_{ij} = -\lambda \frac{dE}{dw_{ij}}$$

Stochastic Gradient descent for sample n

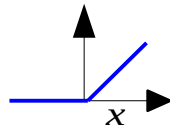
$$\Delta^n w_{ij} = -\lambda \frac{d(^n e)}{dw_{ij}}$$



Multilayer Perceptron

Layers $l = 0 \dots L$ Not power, just layer index

Model $a_i^l = f_i^l \left(\sum_j w_{ij}^l a_j^{l-1} \right)$ $a_i^0 = x_i$

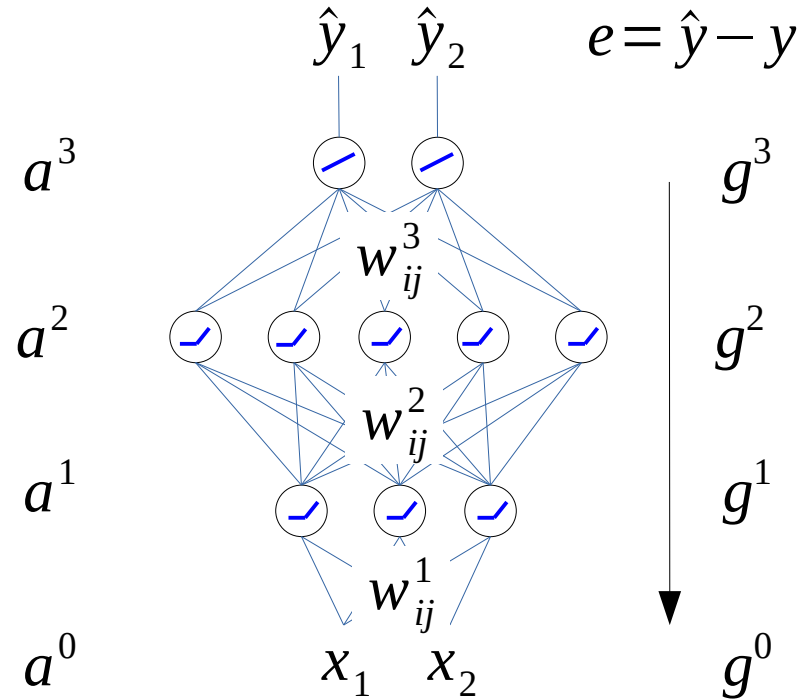
ReLU $a = \max(0, x)$ 

Gradient back propagation

$$g_j^{l-1} = \sum_i f_i^{l'} w_{ij}^l g_i^l \qquad g_i^L = 2 e_i$$

Cost function $E = \sum_n \sum_i |e_i|^2$

Stochastic Gradient descent: $\Delta w_{ij}^l = -\lambda f_i^{l'} g_i^l a_j^{l-1}$



“3 dense network layers” 8

Example: NIST digit classification

Oswaldo Velarde:

Introduce google colab.

Present example in python from

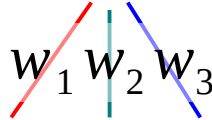
“Deep Learning with Python” Book by François Chollet, 2nd edition, Manning.

Chapter 2, MNIST digit classification with 2 layer dense network.

Convolutional Neural Network (CNN)

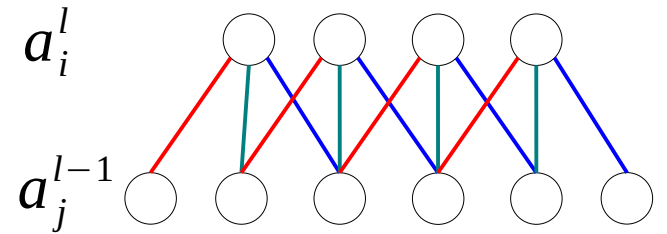
Weights are shift invariant:

$$w_{ij} = w_{i-j} = w_k$$



Model

$$a_i^l = f\left(\sum_k w_k a_{i-k}^{l-1}\right)$$



i.e. all units have the same weights; weights are “tied”

Gradient update is summed over all “tied” weights*

“convolution layer”

$$\Delta w_k^l = -\lambda \sum_i f_i^{l'} g_i^l a_{i-k}^{l-1}$$

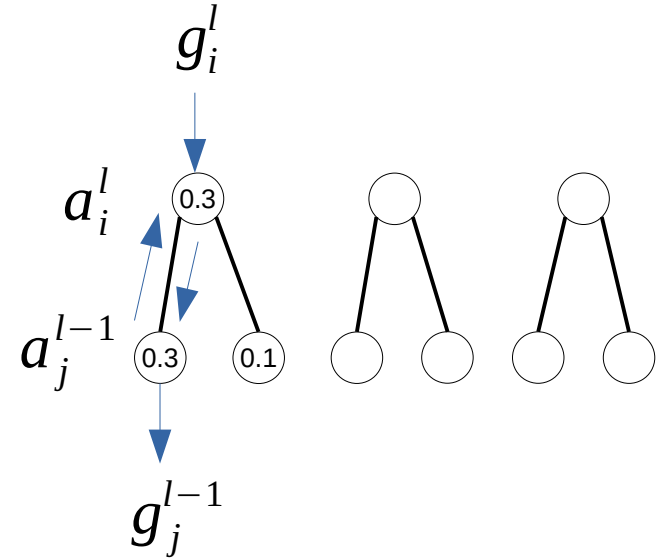
* this is true for any kind of weight symmetry that shares weights

Pooling

Reduce resolution by taking the maximum or mean, e.g.

$$a_i^l = \max(a_i^{l-1}, a_{i-1}^{l-1})$$

The gradient only propagates to the units that contribute to the pooled value, e.g. gradient is zero for units that were not selected in the maximum.



“pooling layer”

Other forms of reducing resolution with convolution:

Stride – skip intermediate pixels; is the same as weighted average pooling.

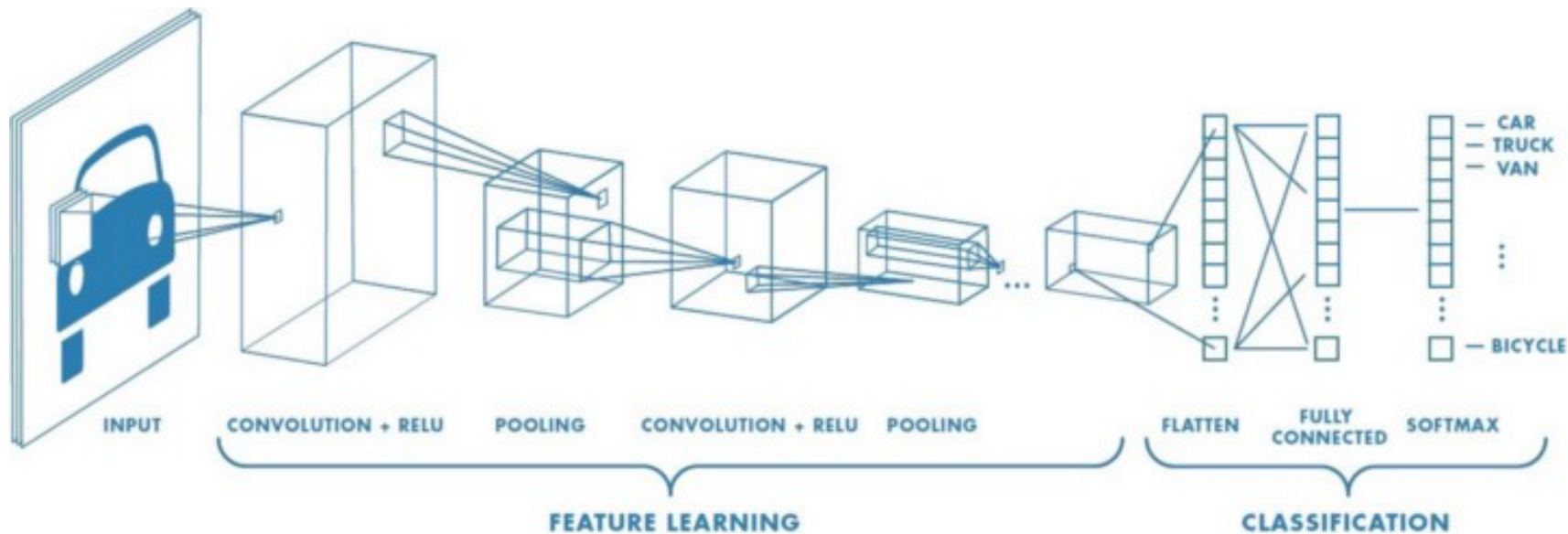
Downsampling – same as weighted average pooling but with predefined weights.

“valid” – reduce size at the boundary, bonus: avoids edge artifacts.



Deep CNN

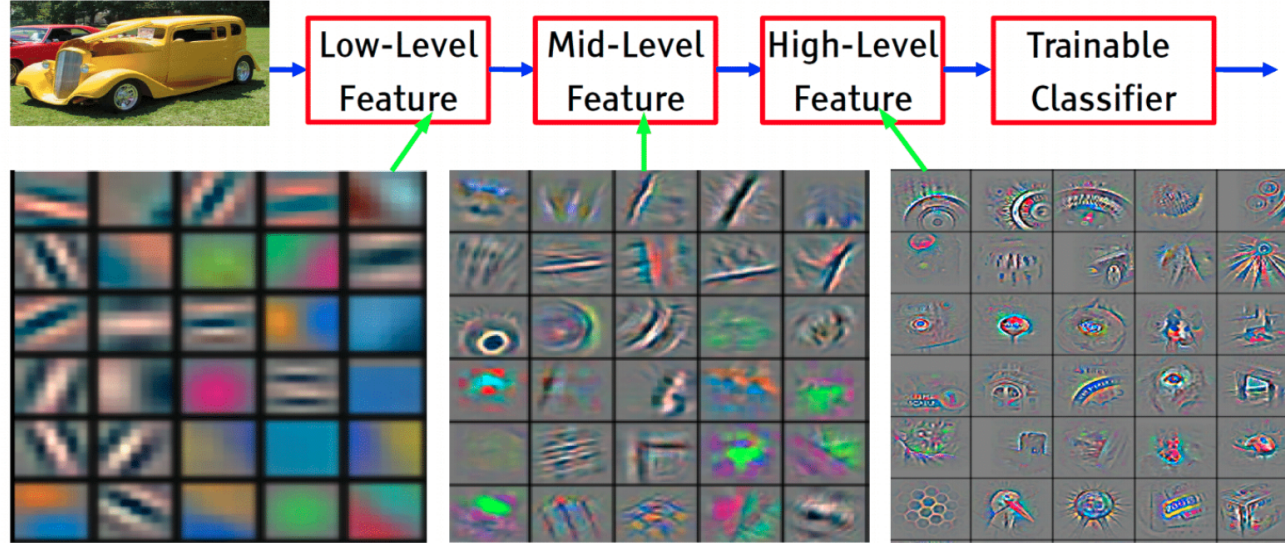
Layers can be stacked up into deep networks. With CNN, typically decreasing spatial resolution while increasing number of “channels”





CNN features

Increasing complexity of features in successive layers.



Deep network tricks

Equations for activation, gradients, and weight updates have all been automated, e.g. in TensorFlow or Pytorch. All that is needed is to define the network architecture, e.g. in Keras.

Deep networks with many layers have a number of problems, but also remarkably effective heuristic solutions (some keywords for independent study):

Repeated multiplication with f' causes vanishing gradients in deep layers. Solutions:

- ReLU non-linearity
- Batch normalization

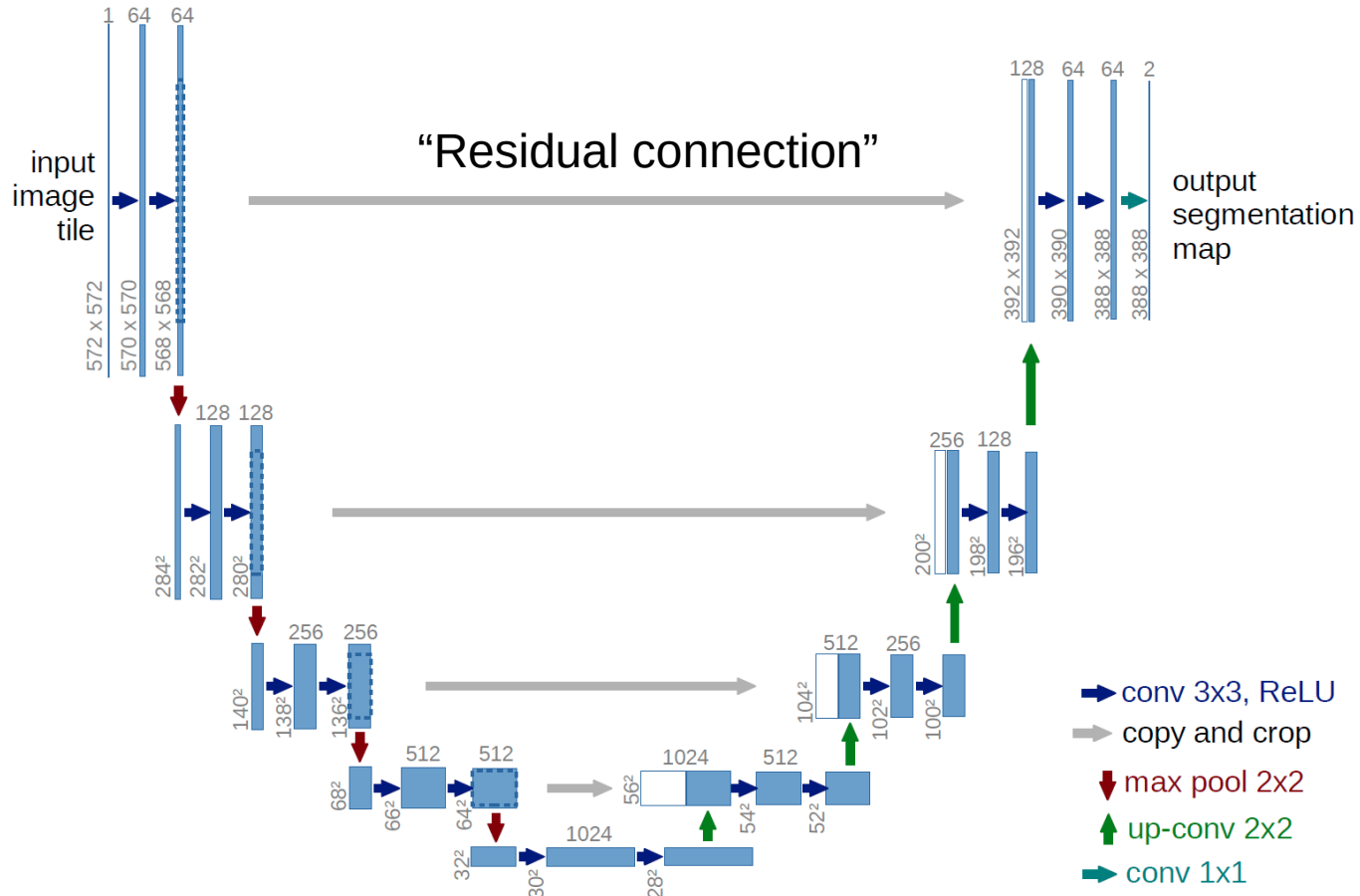
Repeated mixing with $\sum_j w_{ij}$ causes diluted gradients in deep layers (my wording). Solutions:

- Residual connections
- Max pooling
- Attention gating

With millions of free parameters over-training is the norm. Solutions:

- Tying weights (symmetry)
- Dropout
- Early stopping
- Self supervised learning
- Transfer learning

Segmentation with UNets



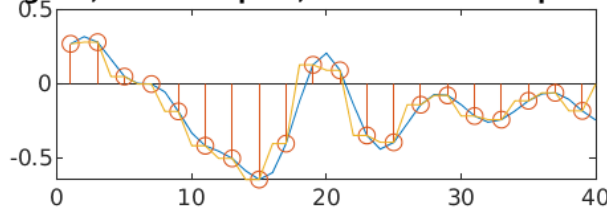
Down and up-sampling

Downsampling by 2: low-pass and then take every other sample, e.g. “mean pooling” with “stride” 2.

Upsampling: insert zeros and then filter with point spread function. “Transpose convolution”

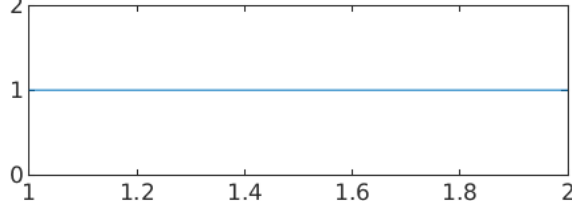


Original, downsampled, and recovered upsampled

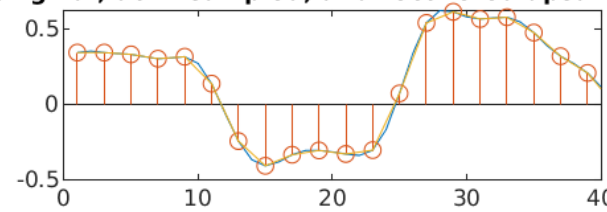


Stride 2
Upsampling 2
kernel 2

upsampling filter, SNR = 21 dB

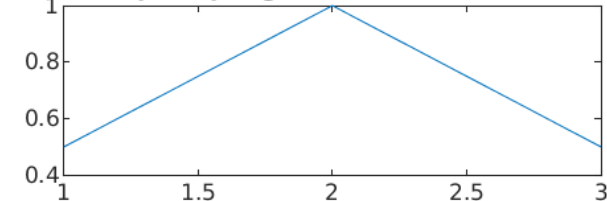


Original, downsampled, and recovered upsampled



Stride 2
Upsampling 2
kernel 3

upsampling filter, SNR = 54 dB



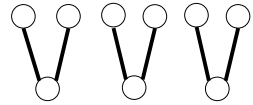
Example: Neuron segmentation

Oswaldo Velarde

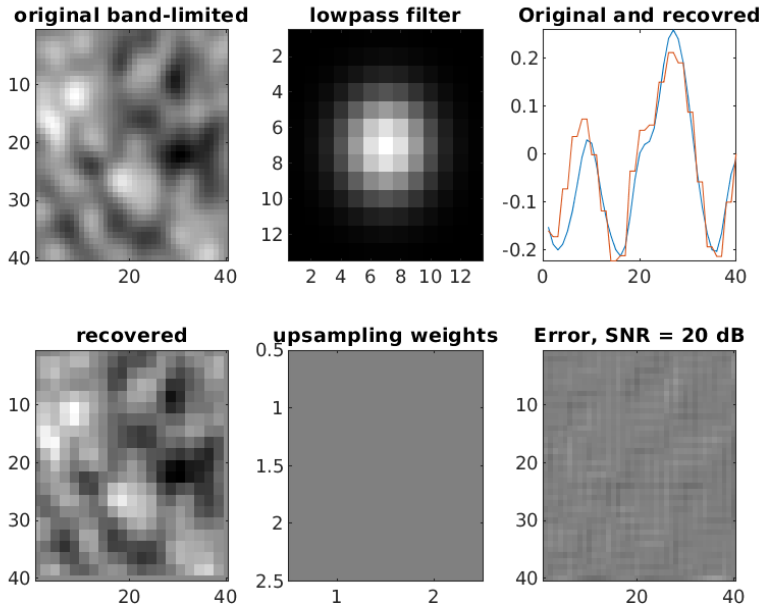
Present example of UNet in colab for segmenting 2D phase contrast images of neurons.

Down and up-sampling

Constant weights

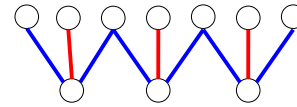


$$w = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

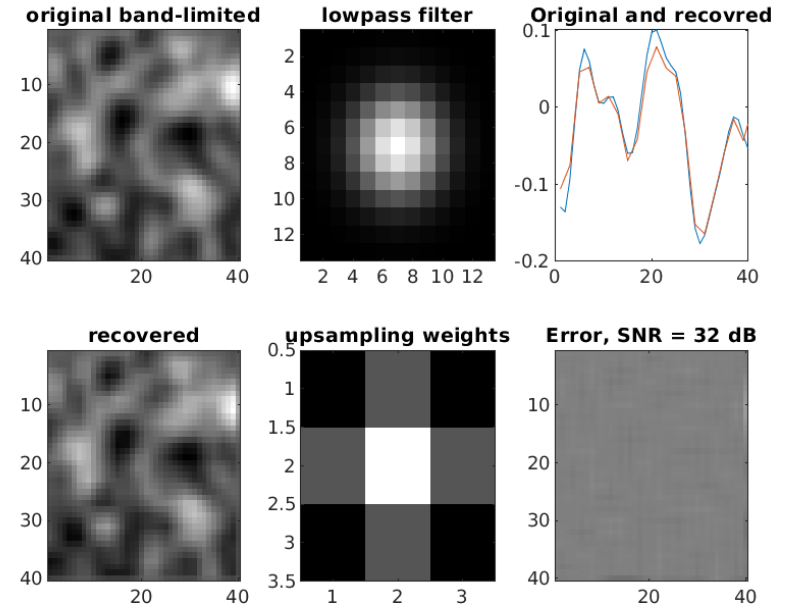


This is TensorFlow default initialization and tends to cause grid-artifacts. Needs to “learn” to avoid them.

Linear weights



$$w = \begin{bmatrix} 0.25 & 0.50 & 0.25 \\ 0.50 & 1.00 & 0.50 \\ 0.25 & 0.50 & 0.25 \end{bmatrix}$$



Optimal without need to learn. Weights can be set fixed.