SPATIAL ECOLOGY

Neural Nets (pt.3), Interpretability and Convolutional Neural Networks

Antonio Fonseca

Agenda

1) Feedforward Neural Networks

- Quick recap
- Extra regularization techniques
- Capacity, Overfitting and Underfitting
- Debugging tips
- Family of optimizers
- Tutorial: more features and different optimizers

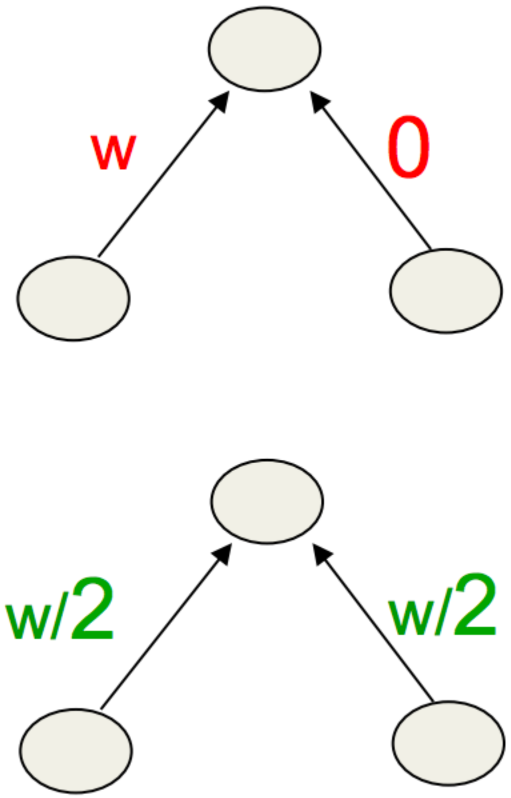
2) Interpretability in Neural Nets

- SHAP and saliency maps
- Tutorial: inspect the importance of features in the tree height dataset.

3) Convolutional Neural Networks

- Kernels, padding, pooling
- Classification tasks
- Tutorial: data batching, classification of satellite images

Regularization

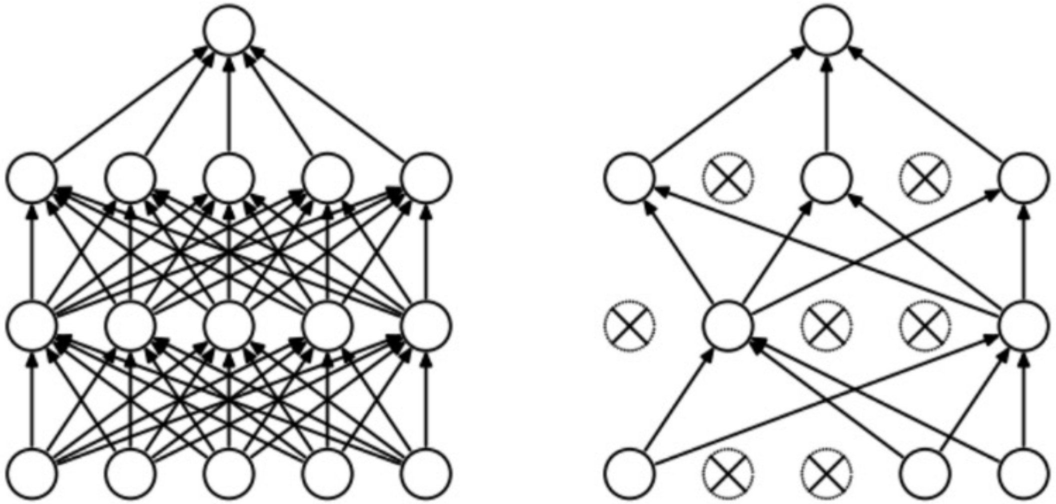


- Prefers to share smaller weights
- Makes model smoother
- More Convex

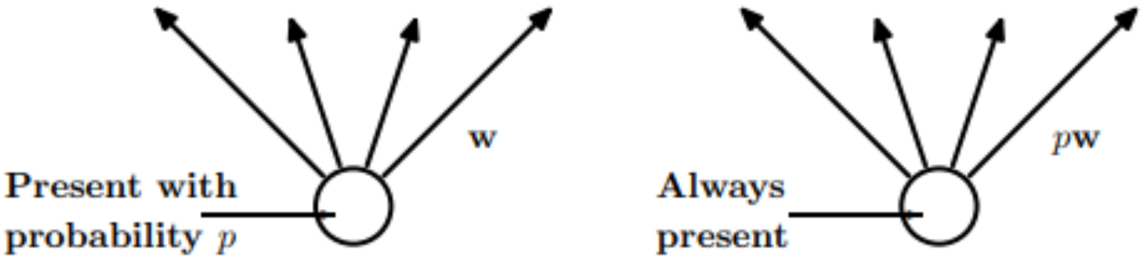
Extra Regularization for Neural Nets

Dropout: accuracy in the absence of certain information

- Prevent dependence on any one (or any small combination) of neurons



(a) Standard Neural Net (b) After applying dropout.



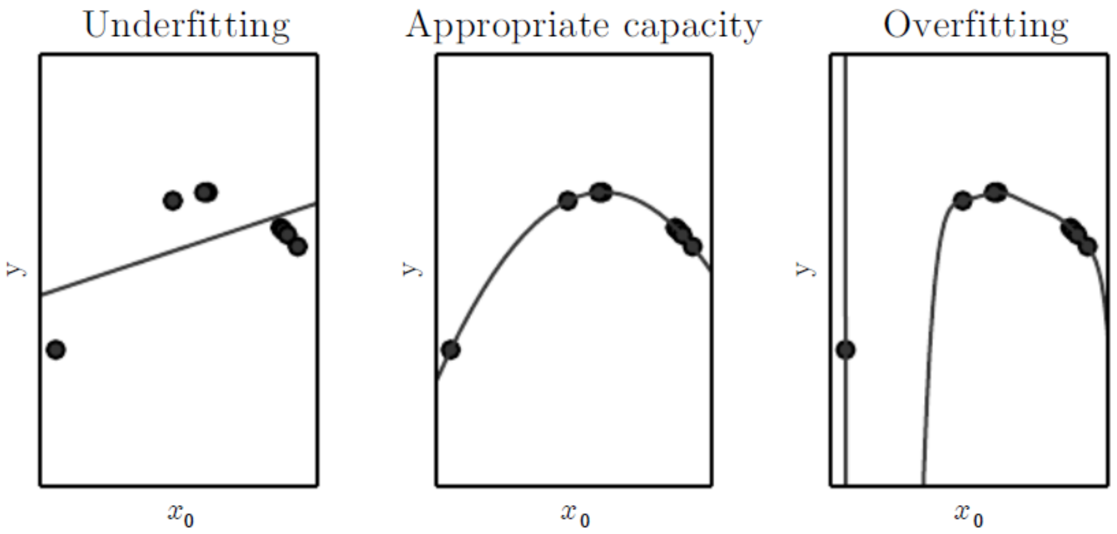
(a) At training time (b) At test time

Capacity, Overfitting and Underfitting

1) Make training error small
2) Make the gap between training and test error small

Capacity (fitting ability)

Low capacity ← $\hat{y} = \sum_{i}^{1} w_i x^i$ $\quad$ $\hat{y} = \sum_{i}^{2} w_i x^i$ $\quad$ $\hat{y} = \sum_{i}^{9} w_i x^i$ → High capacity



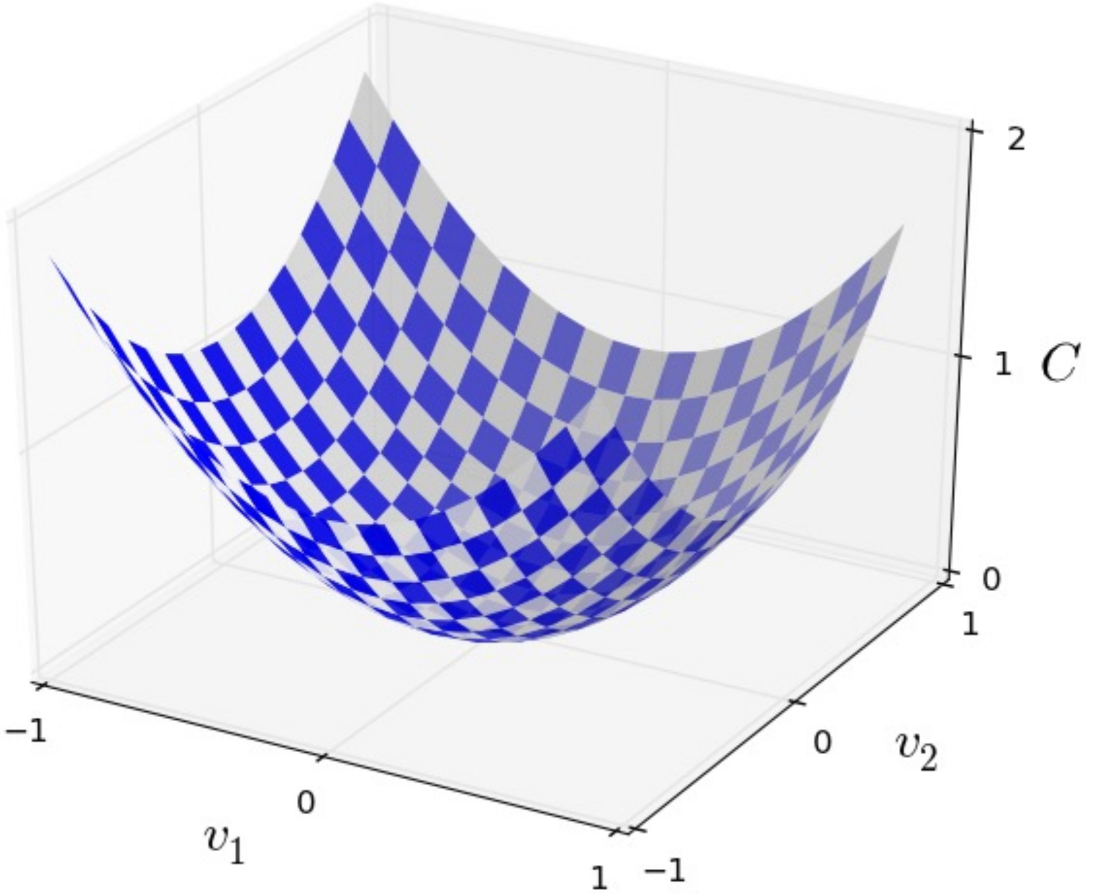
How training works

1. In each ***epoch***, randomly shuffle the training data
2. Partition the shuffled training data into ***mini-batches***
3. For each mini-batch, apply a single step of **gradient descent**
   - **Gradients** are calculated via ***backpropagation***
4. Train for multiple epochs

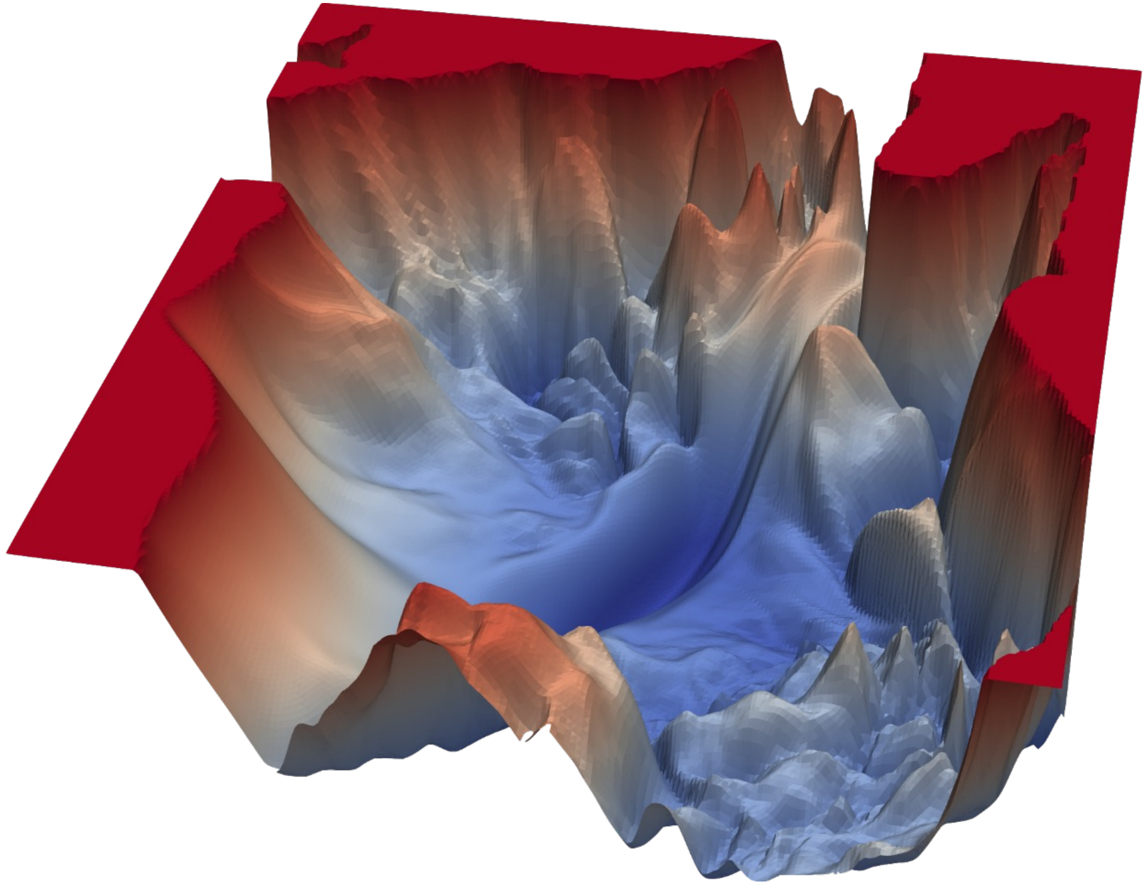
Debugging a neural network

- What can we do?
  - Should we change the learning rate?
  - Should we initialize differently?
  - Do we need more training data?
  - Should we change the architecture?
  - Should we run for more epochs?
  - Are the features relevant for the problem?
- Debugging is an art
  - We'll develop good heuristics for choosing good architectures and hyper parameters

Expectation

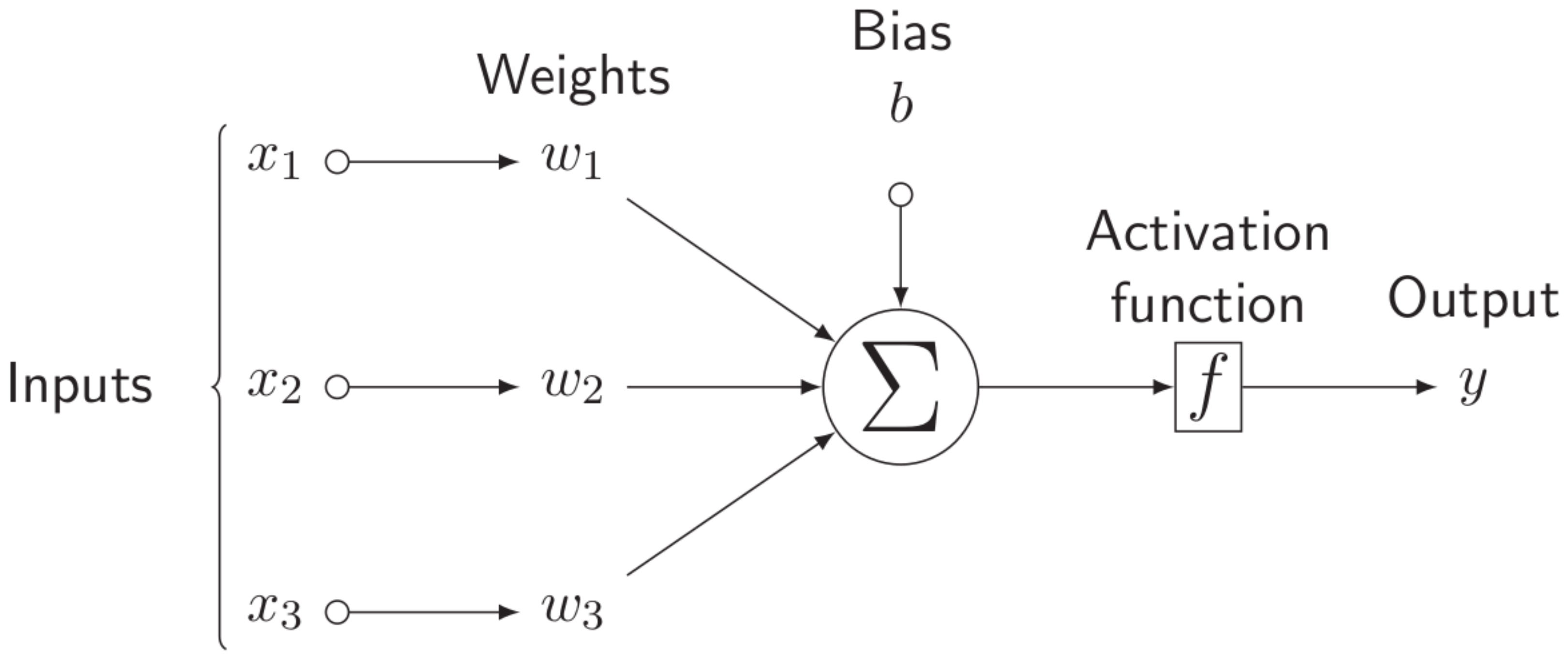


Reality



Perceptron: Threshold Logic

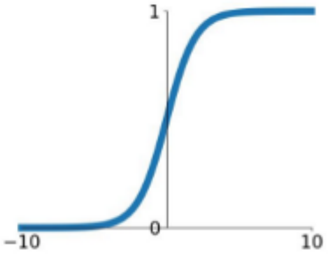
$$\mathcal{L}_{\text{perc}}(\mathbf{x}, y) = \begin{cases} 0 & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) > 0 \\ -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) \leq 0 \end{cases}$$



Activation functions

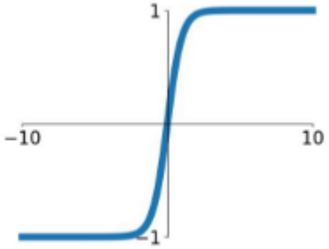
Sigmoid

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



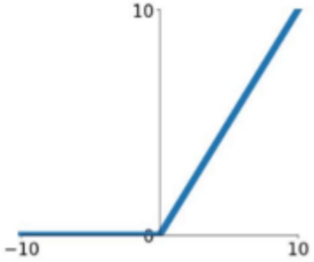
tanh

$\tanh(x)$



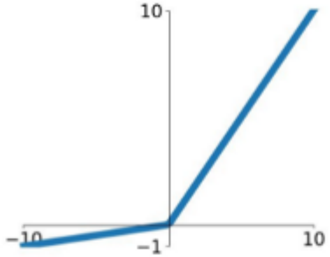
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

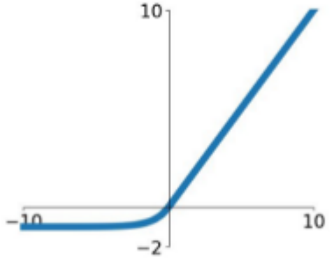


Maxout

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

ELU

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



Optimizers

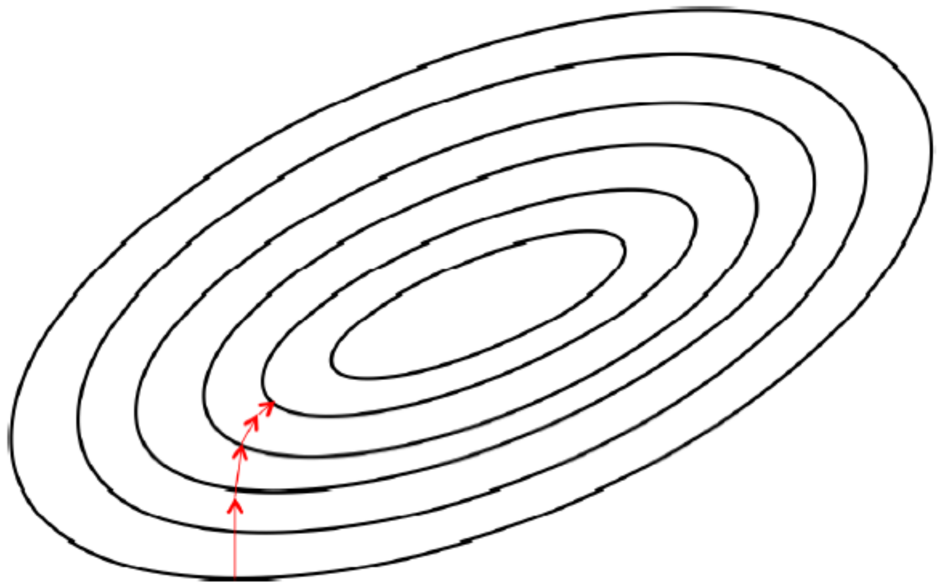
Gradient

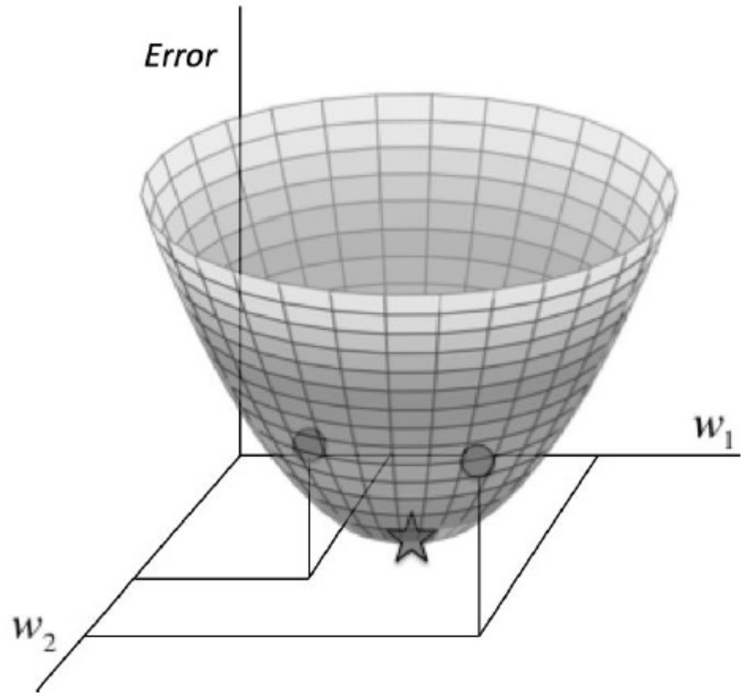
$$\Delta w_k = -\frac{\partial E}{\partial w_k}$$

$$= -\frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$

Stochastic gradient descent (**SGD**)





Optimizers

Practical test:
lr_val = [1; 0.1; 0.01]
momentum_val = 0
nesterov_val = ‘False'
decay_val = 1e-6

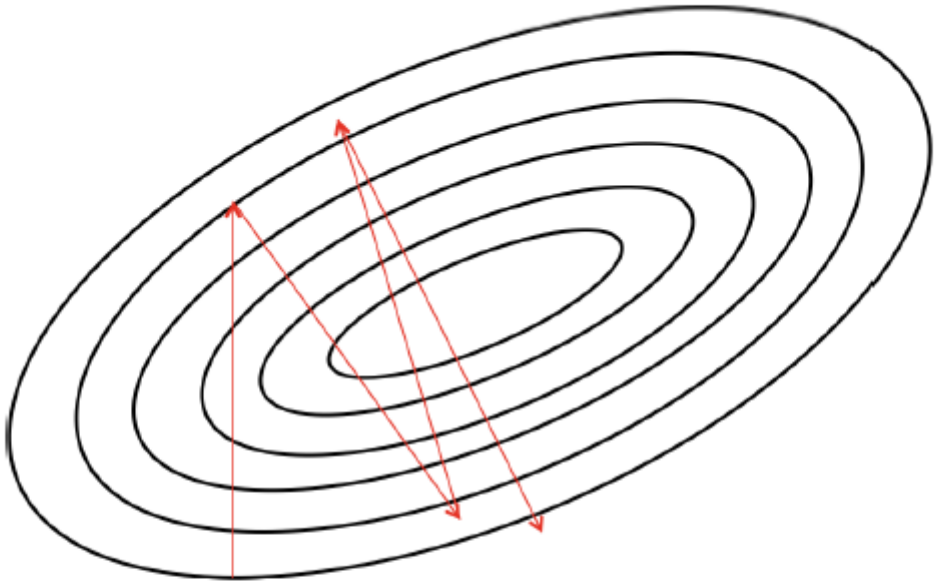
Hyperparameters

- Learning rate (α)

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k}$$

$$= -\alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$



Result of a large learning rate α

Stochastic gradient descent (**SGD**)

Optimizers

Watch out for local minimal areas

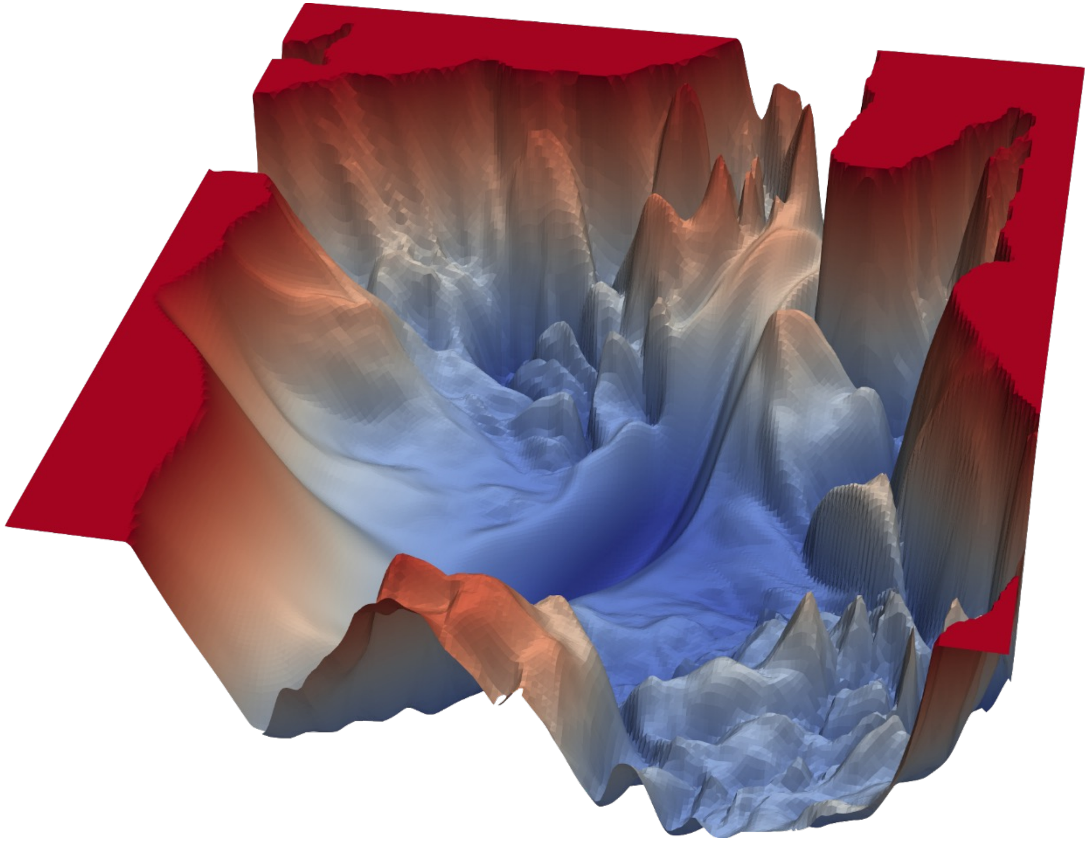
Hyperparameters

- Learning rate (α)

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k}$$

$$= -\alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$



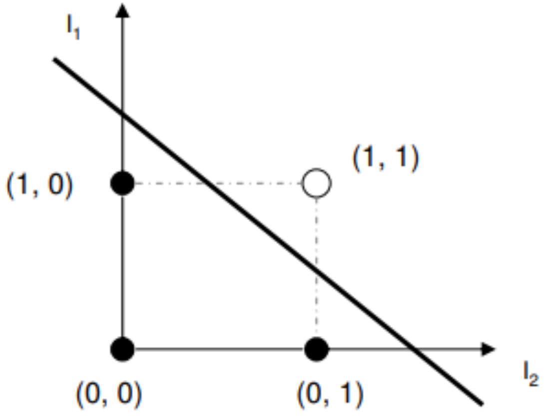
Stochastic gradient descent (**SGD**)

Gradient Descent

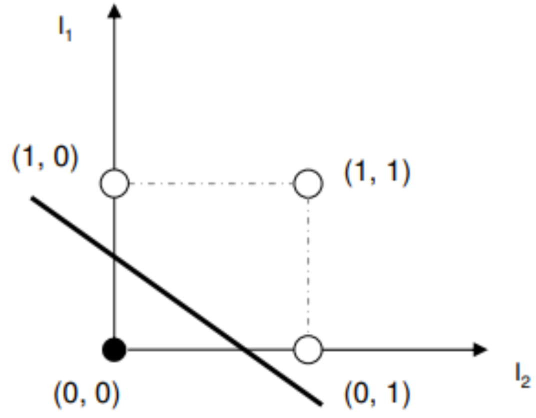
- Gradient descent refers to taking a step in the direction of the ***gradient (partial derivative)*** of a weight or bias with respect to the cost function
- Gradients are propagated backwards through the network in a process known as ***backpropagation***
- The size of the step taken in the direction of the gradient is called the ***learning rate***

Limitations of the Perceptron

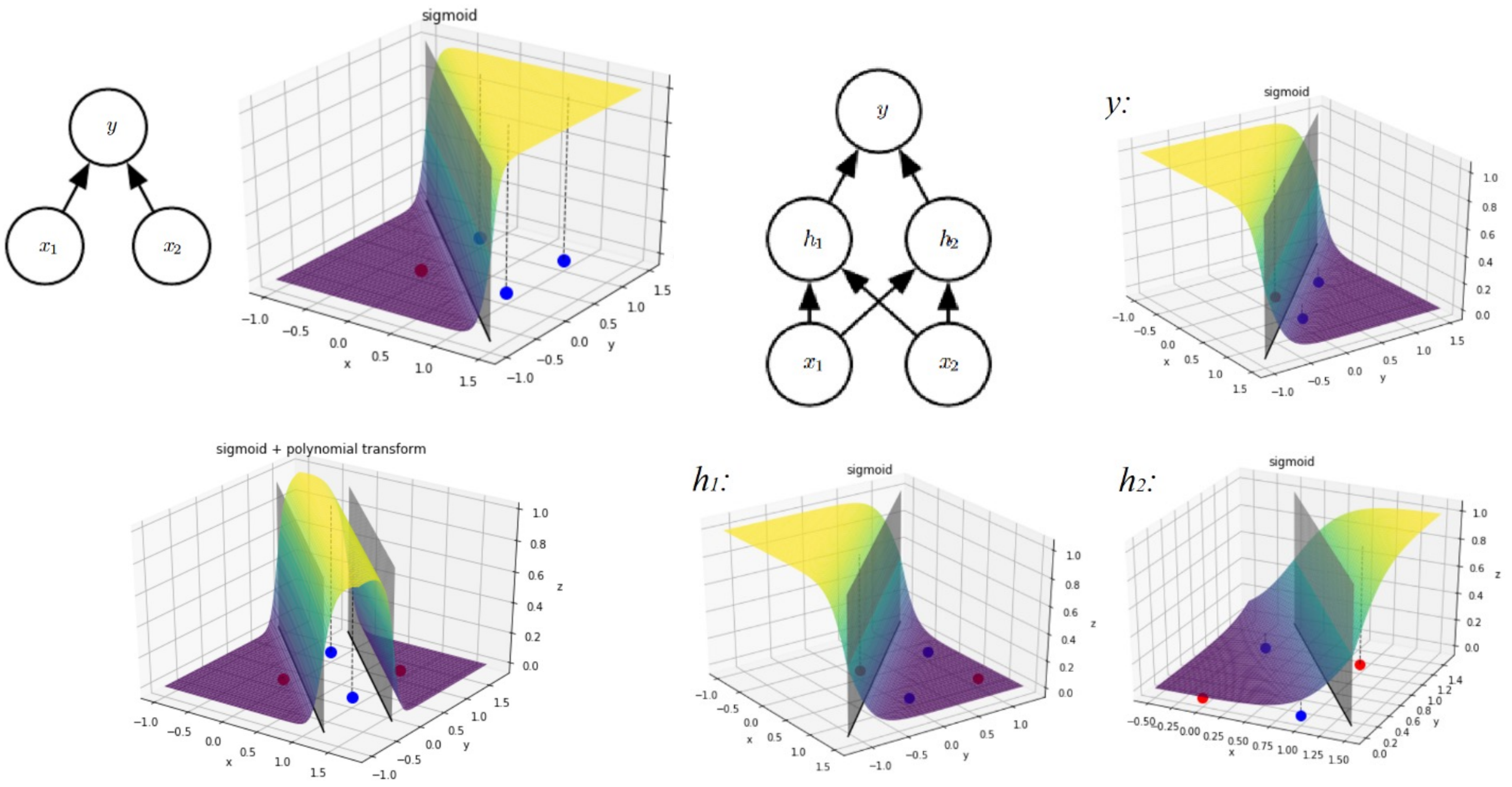
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1

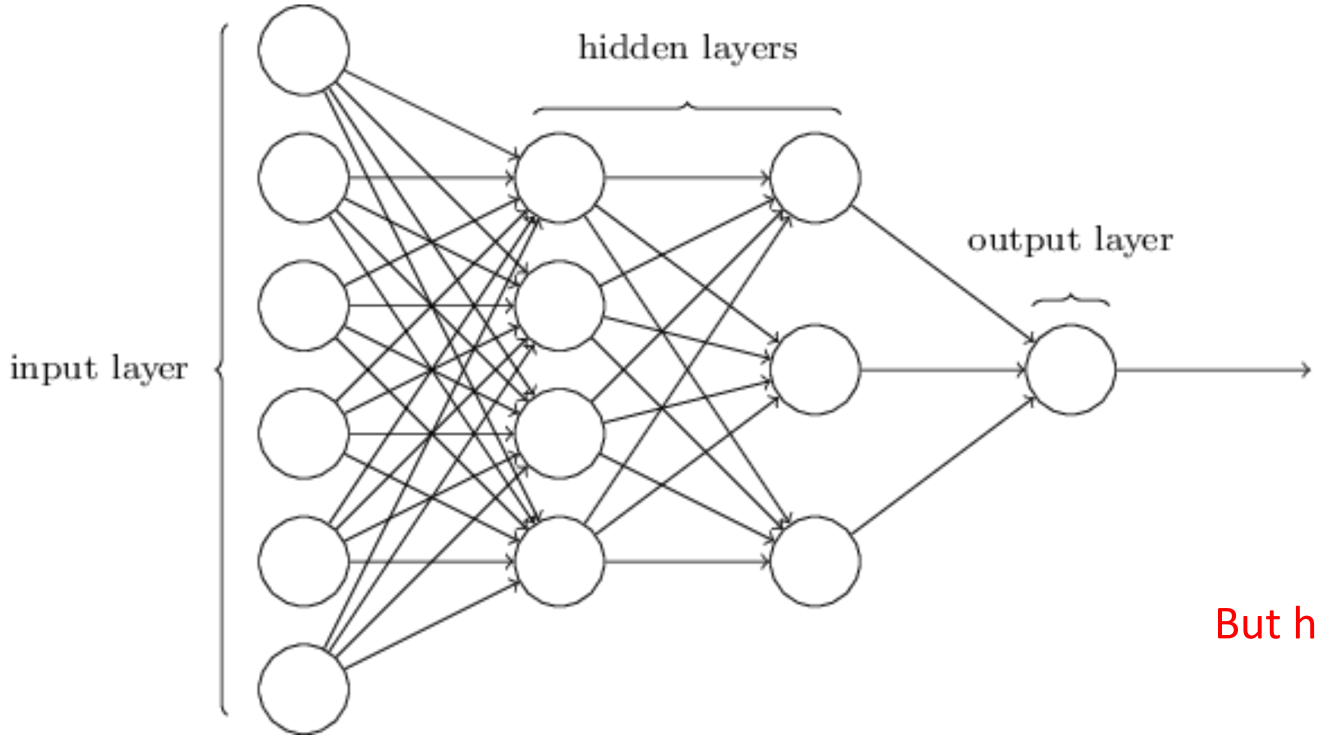


Perceptron



sigmoid
sigmoid + polynomial transform
y:
sigmoid
h1:
sigmoid
h2:
sigmoid

Architecture of Neural Networks

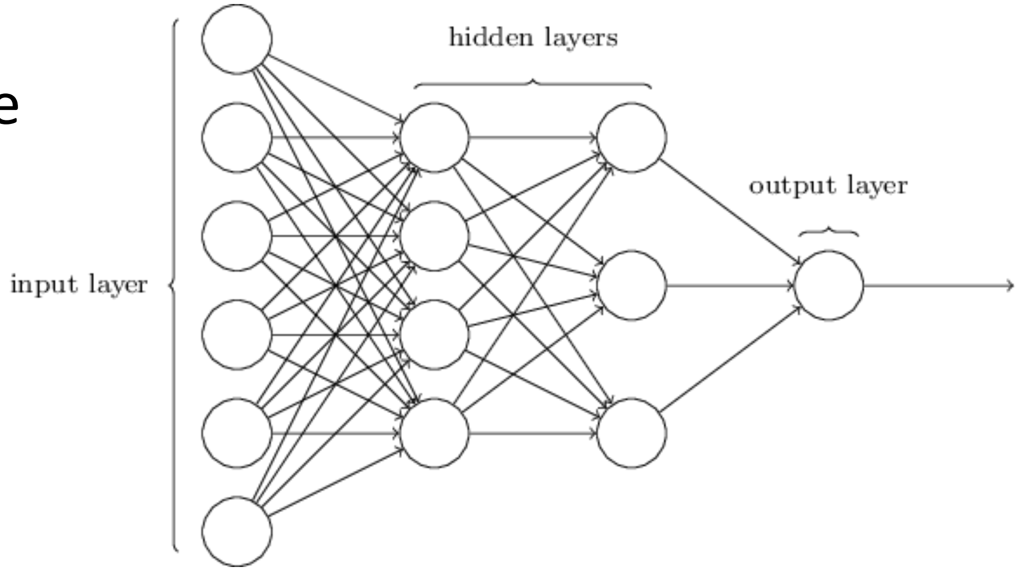


But how do we train it?

- Sometimes called multi-layer perceptron (MLP)
- Output from one layer is used as input for the next (Feedforward network)

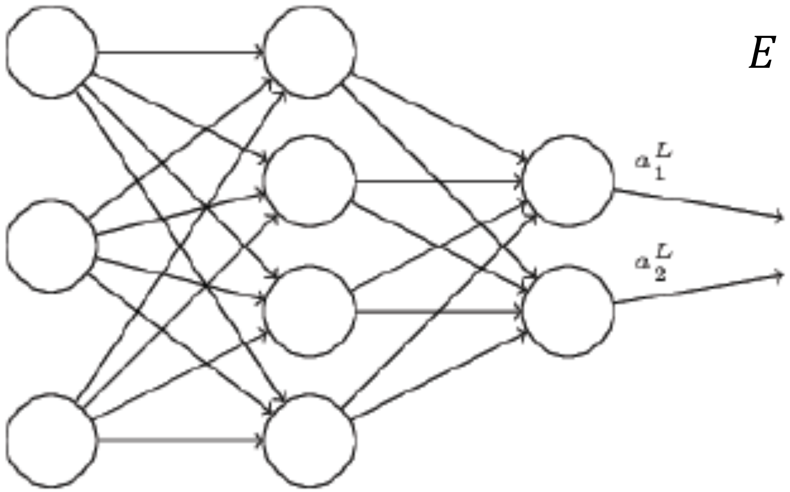
Forward Propagation

- Store weights and biases as matrices
- Suppose we are considering the weights from the second (hidden) layer to the third (output) layer
  - w is the weight matrix with w_{ji} the weight for the connection between the ith neuron in the second layer and the jth neuron in the third layer
  - b is the vector of biases in the third layer
  - a is the vector of activations (output) of the 2nd layer
  - a' the vector of activations (output) of the third layer



$$\boldsymbol{a}' = \boldsymbol{\sigma}(\boldsymbol{w}\boldsymbol{a} + \boldsymbol{b})$$

Backpropagation



$$E = \frac{1}{2}\sum_i \left(a_i^L - y_i\right)^2$$

$$cost\ E = E(a^L)$$

1. **Input x:** Set the corresponding activation a^1 for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

$$z_j^l = \sum_i w_{ji}^l a_i^{l-1} + b_j^l \qquad a_j^l = \sigma\left(\sum_i w_{ji}^l a_i^{l-1} + b_j^l\right) = \sigma(z_j^l)$$

$$\delta_j^L \equiv \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L}\sigma'(z_j^L) \qquad (1)$$

$$\delta_j^l \equiv \frac{\partial E}{\partial z_j^l} = \frac{\partial E}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial z_j^l} = \frac{\partial z_i^{l+1}}{\partial z_j^l}\delta_i^{l+1}$$

$$= \frac{\partial\left(\sum_i w_{ij}^{l+1} a_j^l + b_i^{l+1}\right)}{\partial z_j^l}\delta_j^{l+1} = \sum_i w_{ij}^{l+1}\delta_i^{l+1}\sigma'(z_j^l) \qquad (2)$$

$$\frac{\partial E}{\partial w_{ji}^l} = \frac{\partial E}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_j^l}\frac{\partial (w_{ji}^l a_i^{l-1})}{\partial w_{ji}^l}$$

Optimizers

Hyperparameters

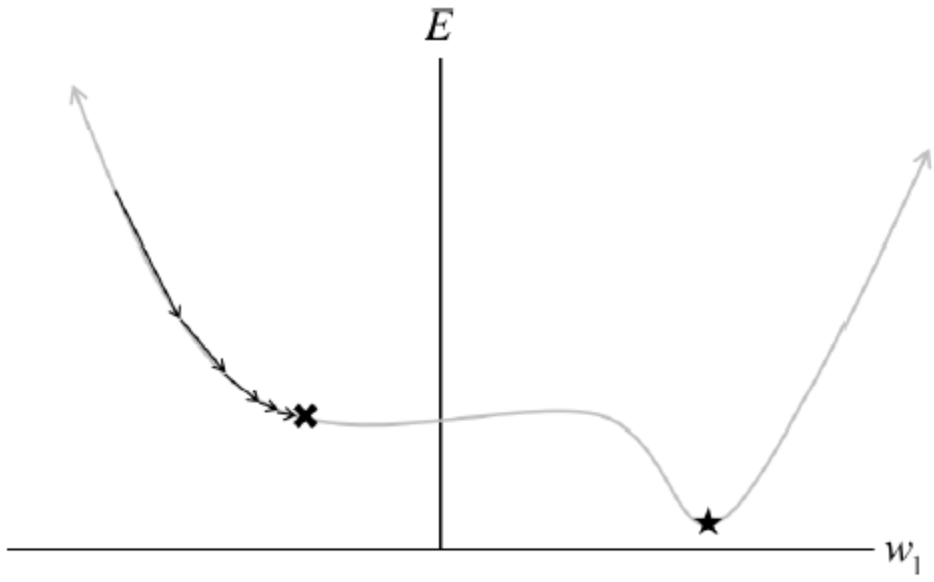
- Learning rate (α)

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k}$$

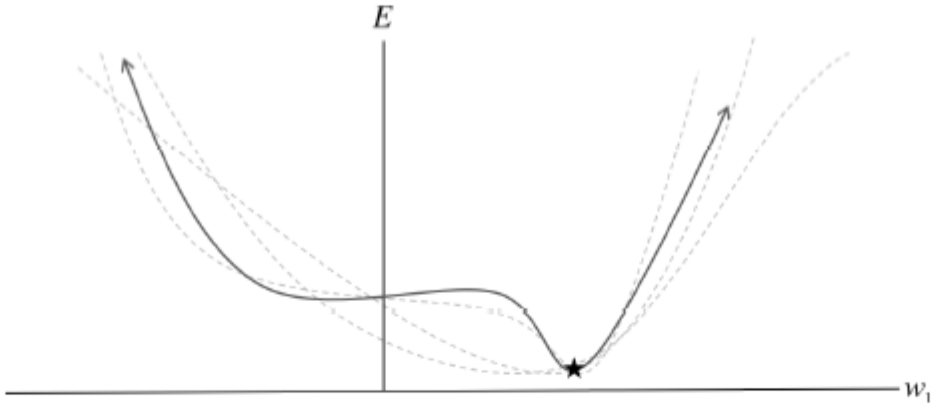
$$= -\alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + \Delta w_k$$

Stochastic gradient descent (**SGD**)



Local Minima



Multiple samples

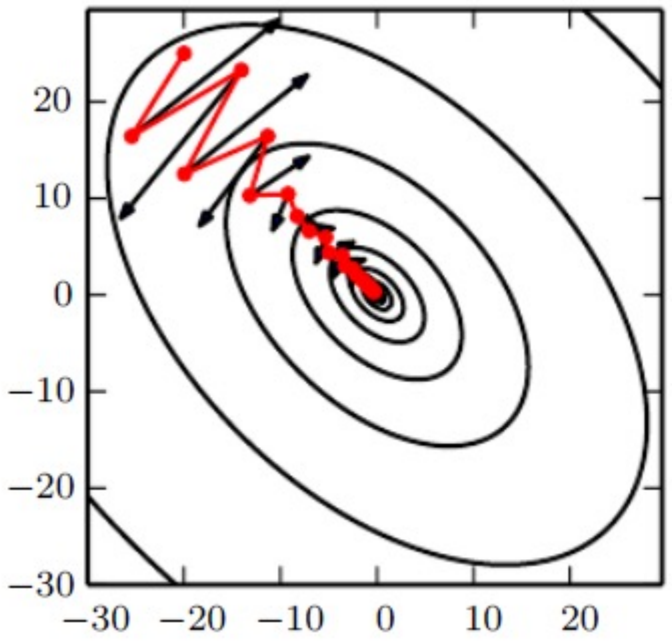
Optimizers

Hyperparameters

- Learning rate (α)
- Momentum (β)

$$v_{i+1} = v\beta - \alpha \frac{\partial}{\partial w_k}\left(\frac{1}{m}\sum_i (w^T X_i - y_i)_i^2\right)$$

$$w_{i+1} = w_i + v$$



SGD

SGD+Momentum

Stochastic gradient descent with momentum (**SGD+Momentum**)

Optimizers

Hard to pick right hyperparameters

- Small learning rate: long convergence time
- Large learning rate: convergence problems

Adagrad: adapts learning rate to each parameter

$$\Delta w_{k,t} = -\alpha \frac{\partial E_t}{\partial w_{k,t}} = -\alpha \nabla_w E(w_t)$$

- Learning rate might decrease too fast
- Might not converge

$$g_{t,i} = \nabla_w E(w_{t,i})$$

$$G_{t+1,i} = G_{t,i} + g_{t,i} \odot g_{t,i}$$

$$\Longrightarrow \quad w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

Optimizers

RMSprop: decaying average of the past squared gradients

Adadelta

$$E[\Delta_w^2]_t = \gamma E[\Delta_w^2]_{t-1} + (1-\gamma)\Delta_w^2$$

$$\Delta w_t = \frac{\sqrt{E[\Delta_w^2]_t + \epsilon}}{\sqrt{G_{t,i} + \epsilon}} g_t$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

Decaying average

$$\Delta w_{k,t} = -\alpha \frac{\partial E_t}{\partial w_{k,t}} = -\alpha \nabla_w E(w_t) = -\alpha g_{t,i}$$

$$g_{t,i} = \nabla_w E(w_{t,i})$$

$$G_{t+1,i} = \gamma G_{t,i} + (1-\gamma) g_{t,i} \odot g_{t,i}$$

$$\Rightarrow \quad w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

Optimizers

ADAM: decaying average of the past squared gradients and momentum

RMSprop / Adadelta

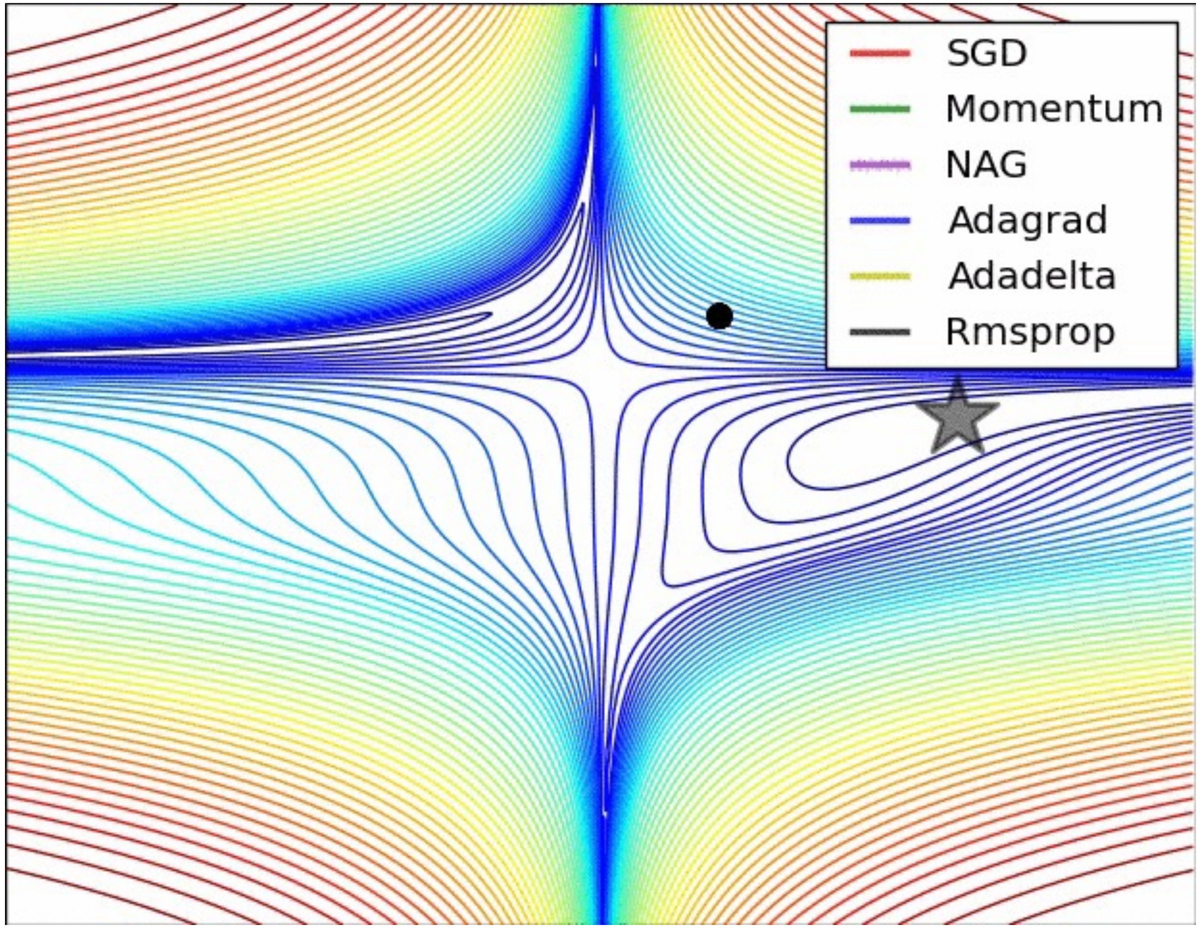
$$g_{t,i} = \nabla_w E(w_{t,i})$$

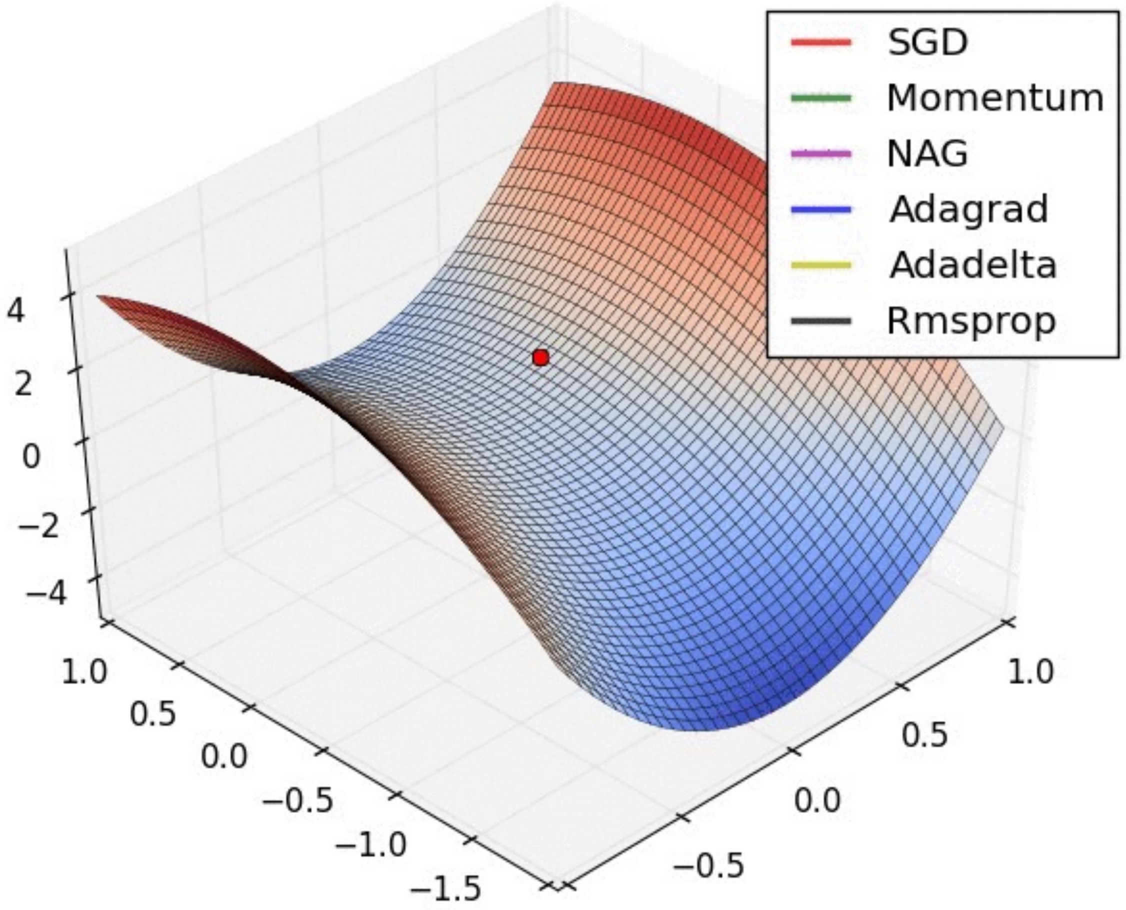
$$G_{t+1,i} = \gamma G_{t,i} + (1-\gamma) g_{t,i} \odot g_{t,i}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$w_{t+1,i} = w_{t,i} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$





Which optimizer is the best?

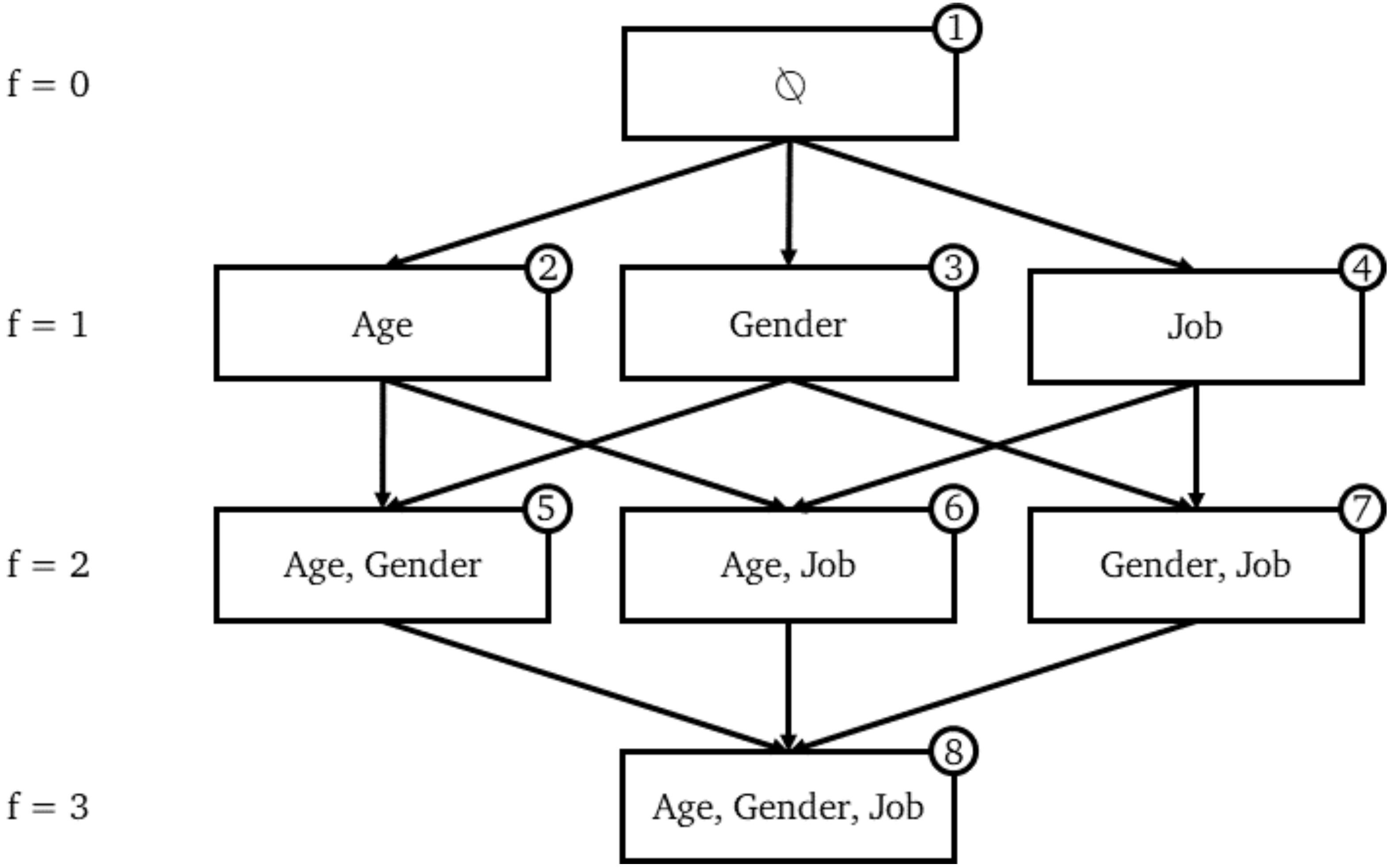
SHAP Values

- SHAP — which stands for SHapley Additive exPlanations
- Reverse-engineer the output of any predictive model
  - Gradient boosting,
  - Neural network,
  - Actually, anything that takes features as input and produces some prediction.

- Approach based on game theory

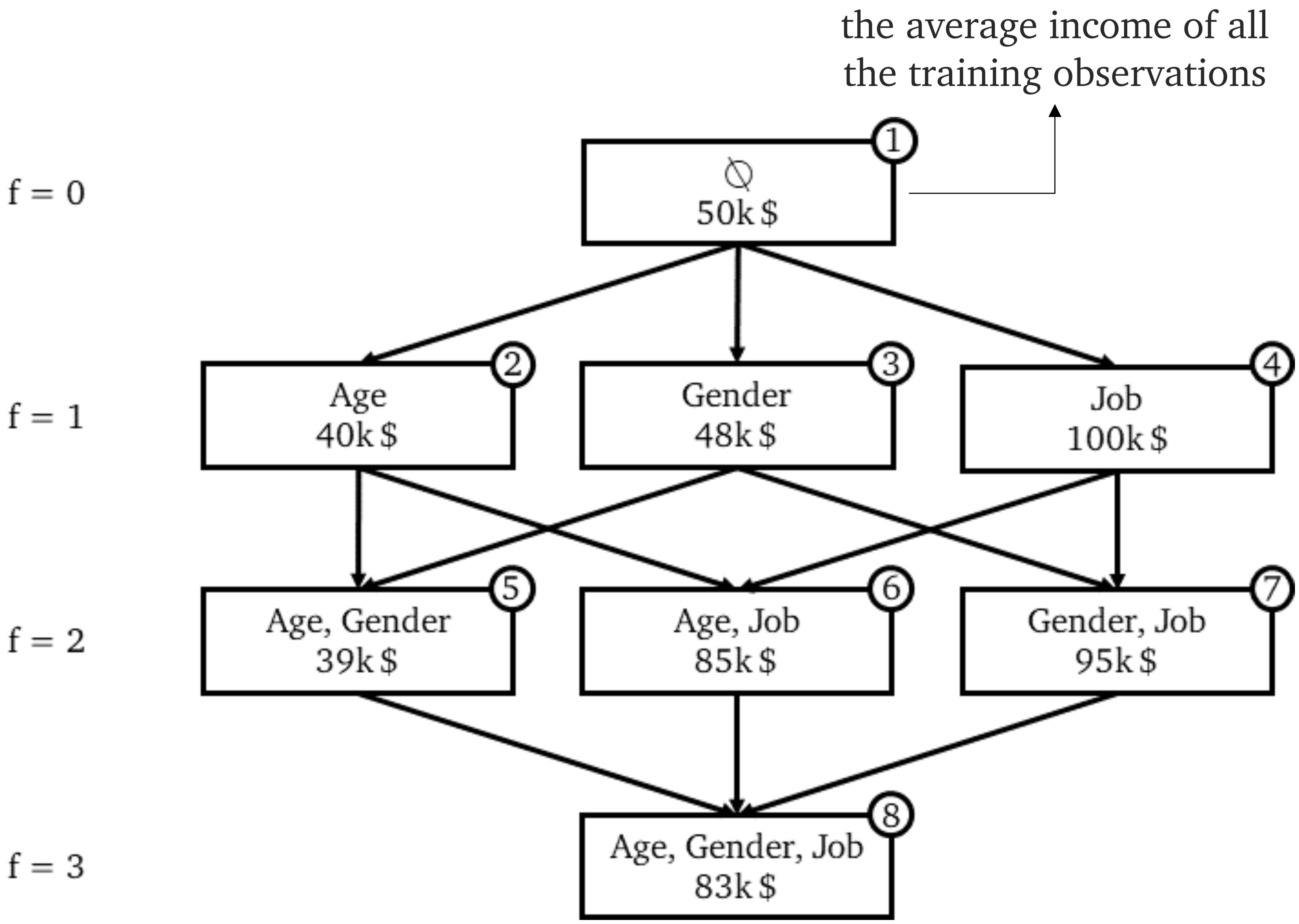
Example

- Regression model that predicts the income of a person knowing age, gender and job of the person.



Example

- Pass new observation through the 8 trained models



Marginal Contribution

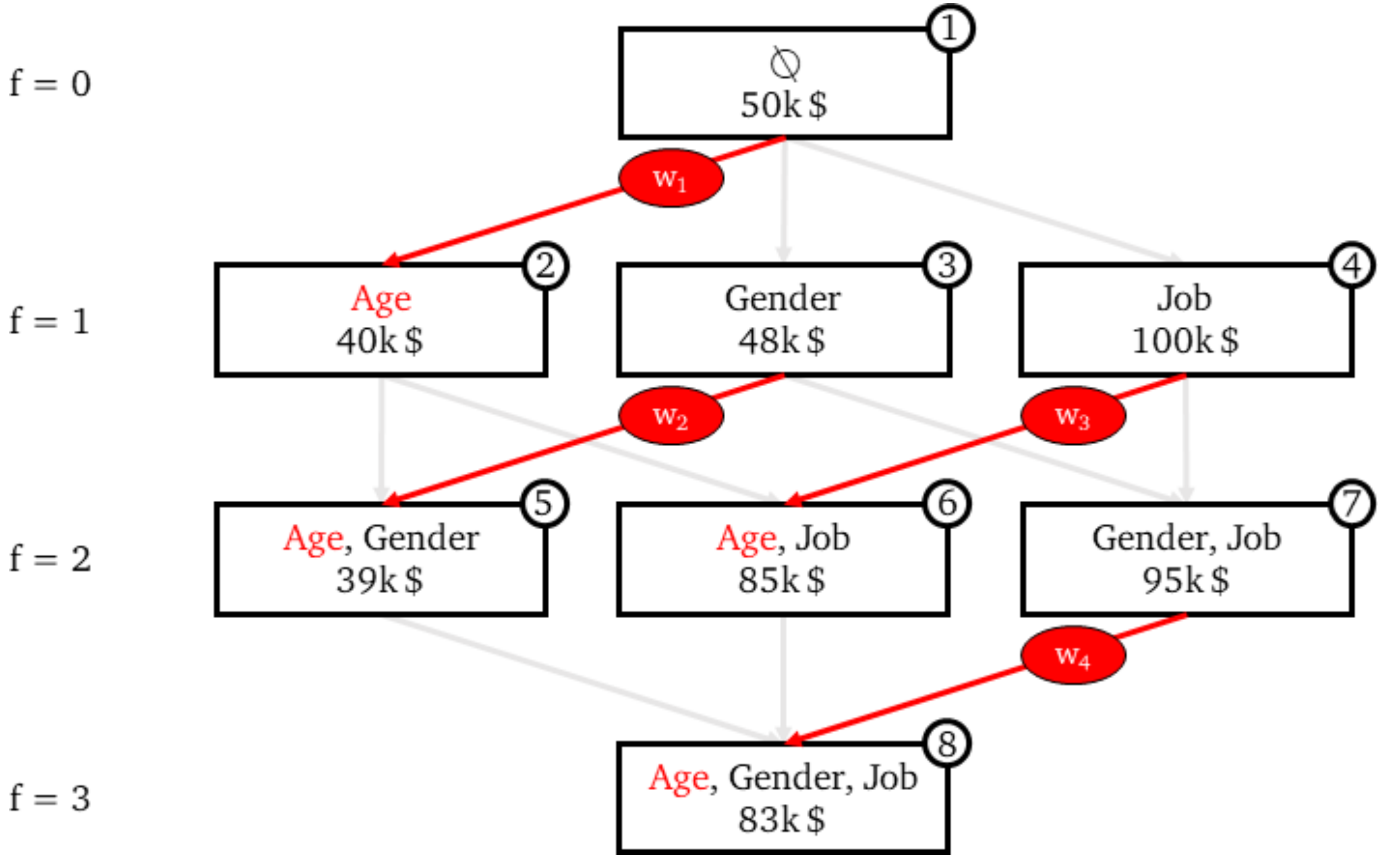
The marginal contribution brought by Age to the model containing only Age as a feature is -10k

$$MC_{Age,\{Age\}}(x_0) = Predict_{\{Age\}}(x_0) - Predict_{\varnothing}(x_0) = 40k\$ - 50k\$ = -10k\$$$

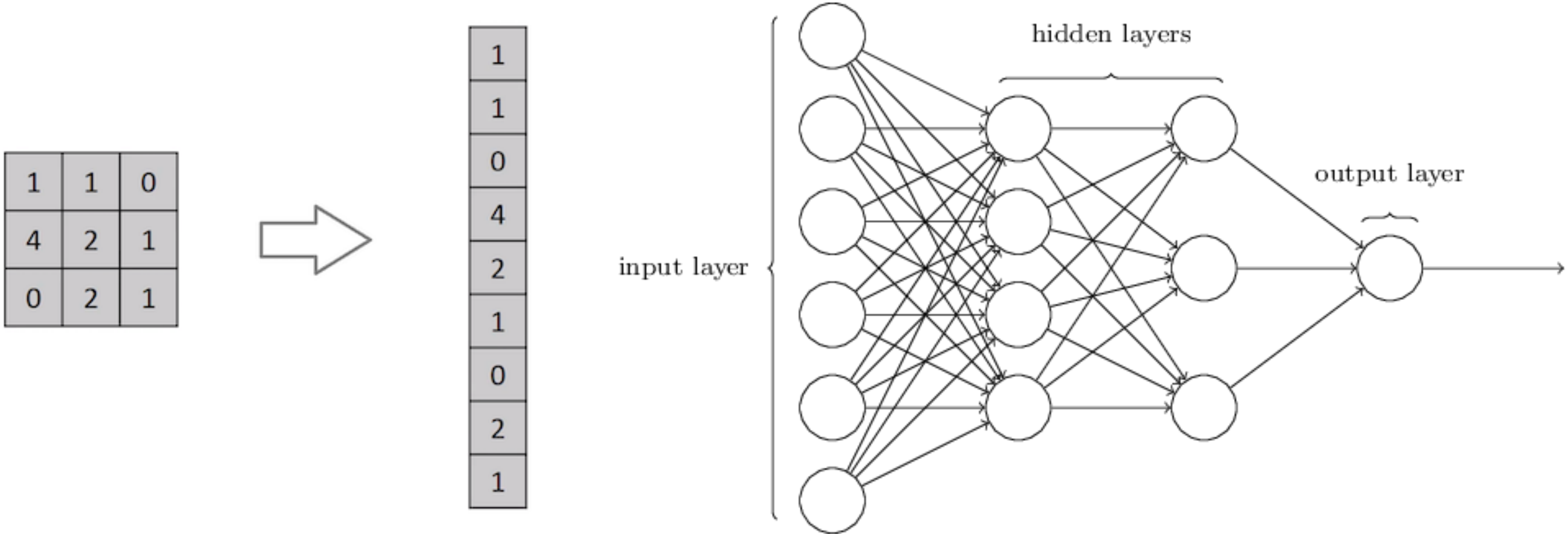
The marginal contribution of Age in all the models where Age is present

$$\begin{aligned} SHAP_{Age}(x_0) = {} & w_1 \times MC_{Age,\{Age\}}(x_0) + \\ & w_2 \times MC_{Age,\{Age,Gender\}}(x_0) + \\ & w_3 \times MC_{Age,\{Age,Job\}}(x_0) + \\ & w_4 \times MC_{Age,\{Age,Gender,Job\}}(x_0) \end{aligned}$$

where $w_1 + w_2 + w_3 + w_4 = 1$.

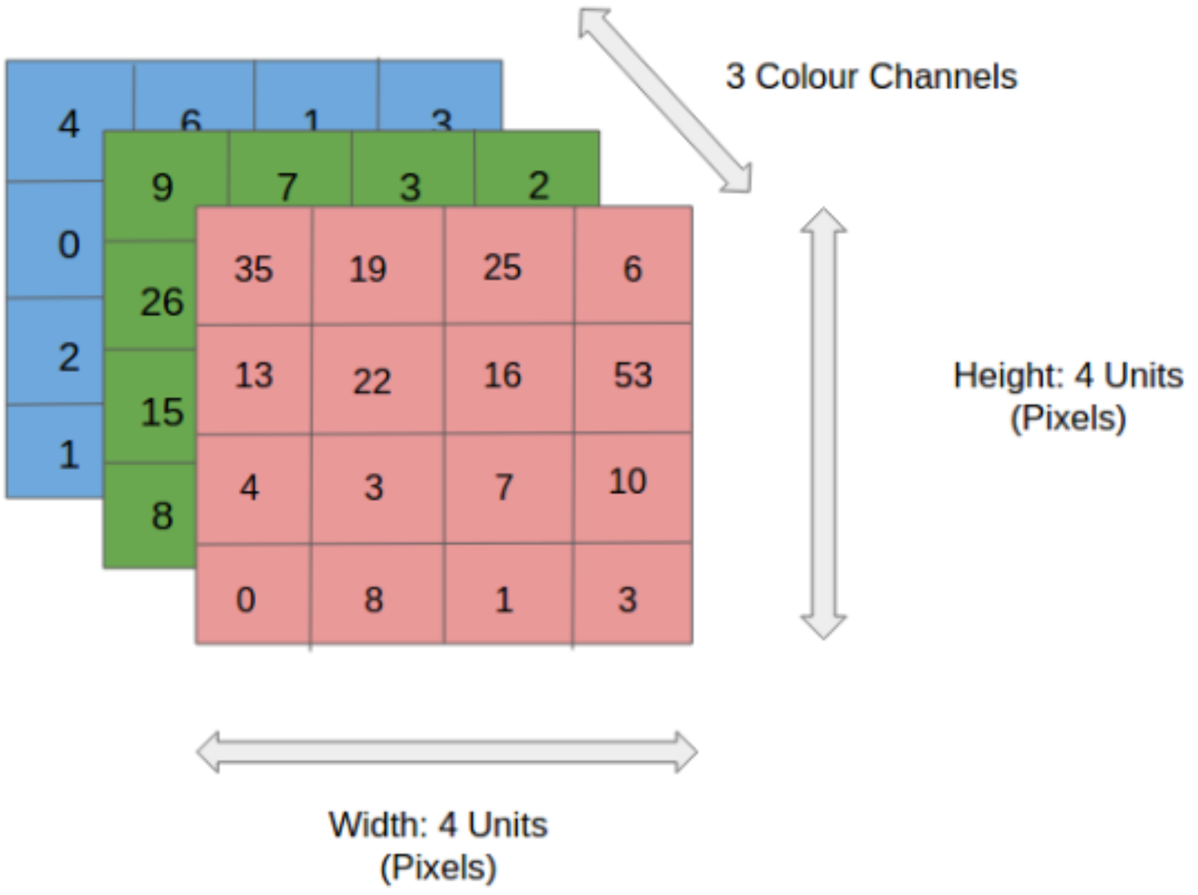


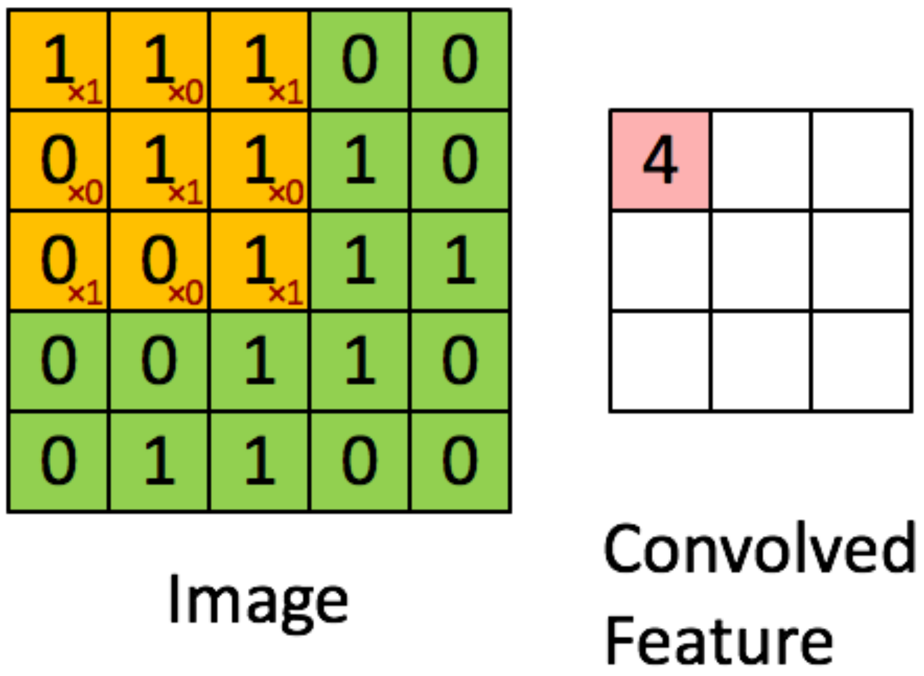
Handling images with Neural Networks



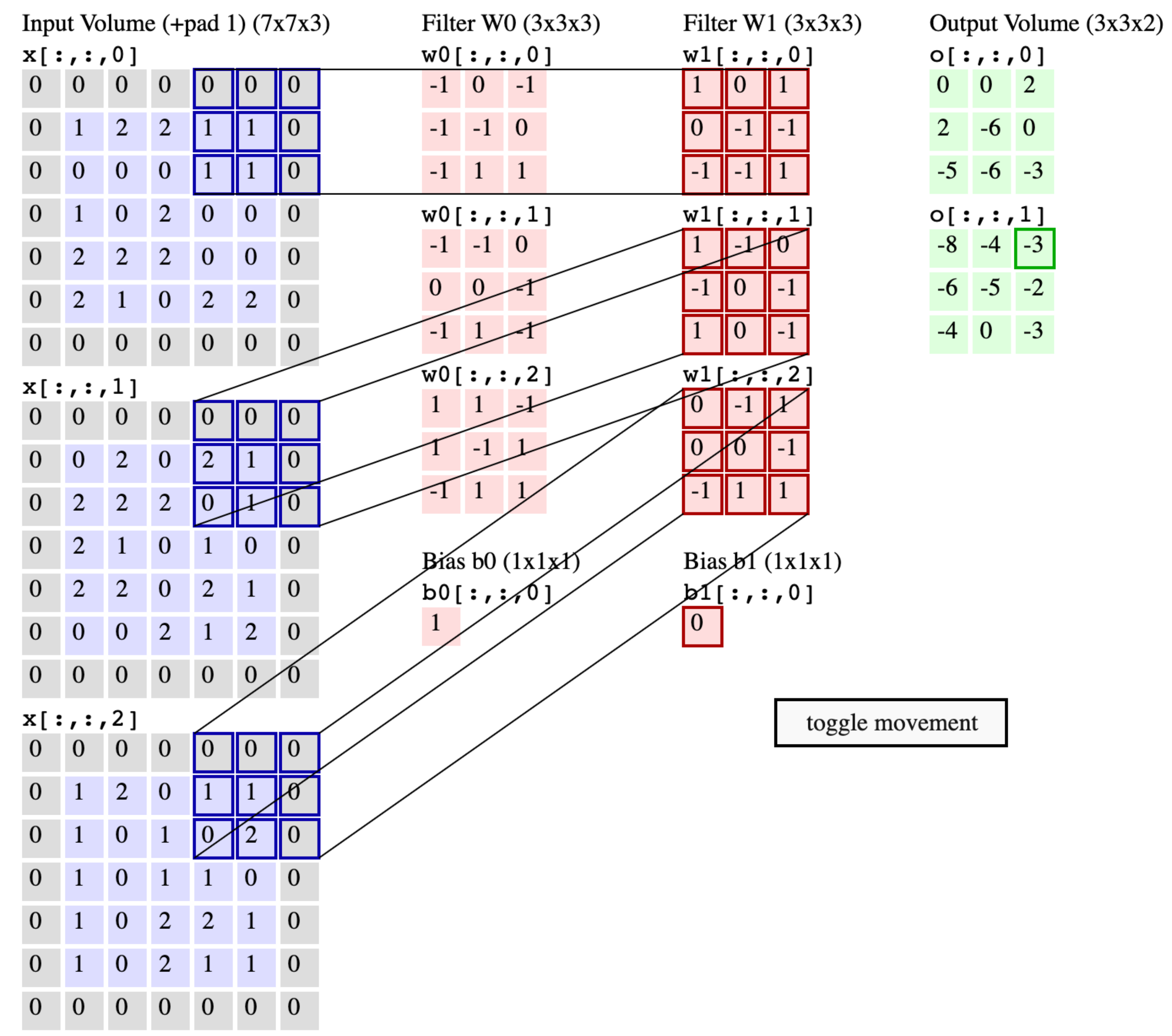
Works well for simple images, but fails when there are more complex patterns in the image

Convolutional Neural Networks

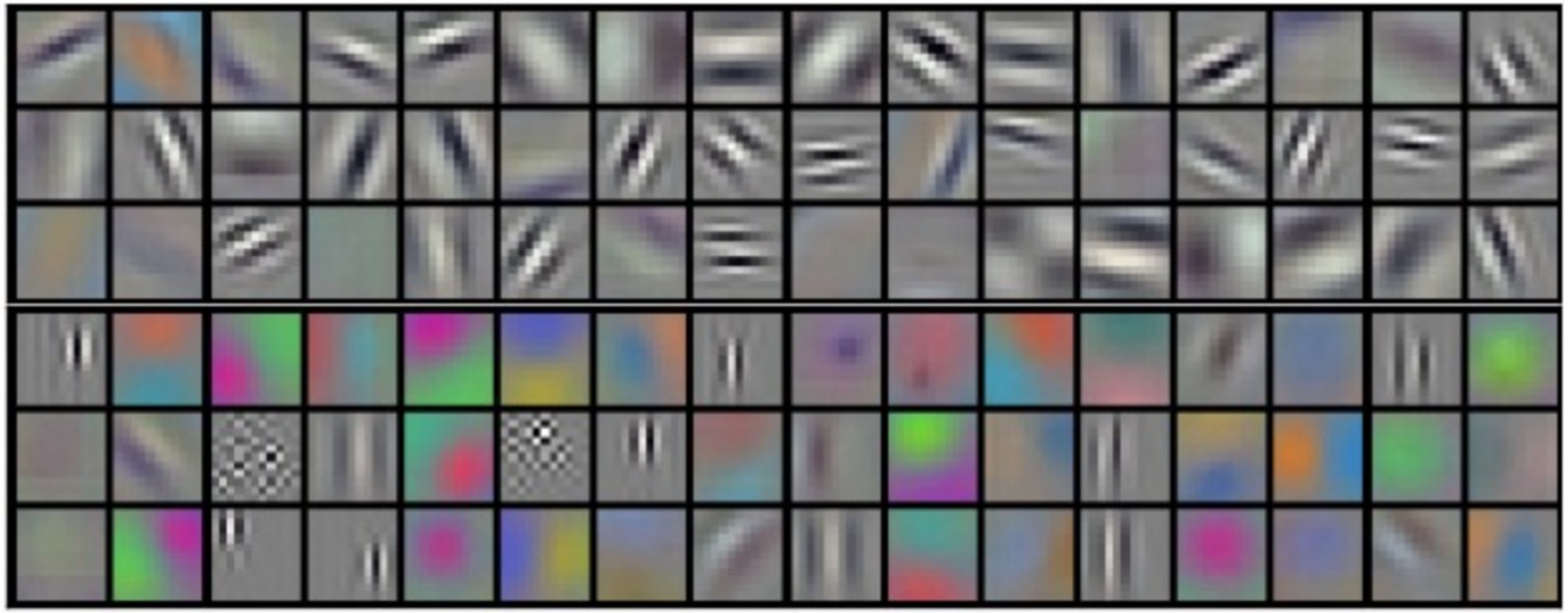




CNN over the image channels

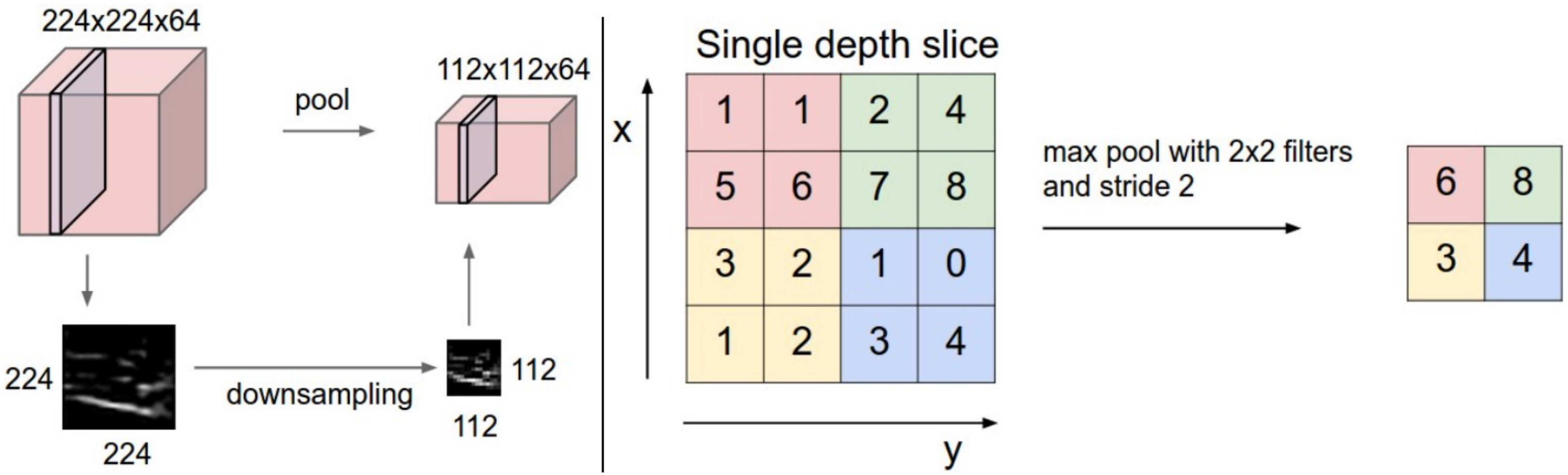


Kernels



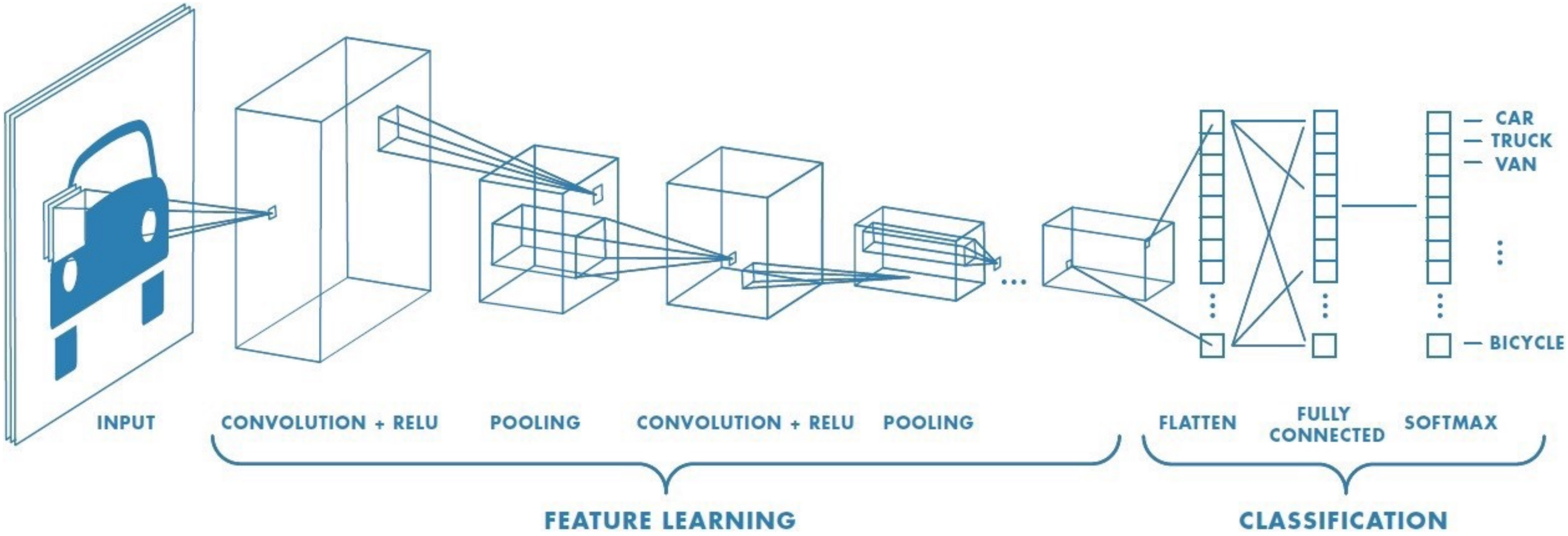
Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Padding and Pooling



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Image Classification



(Putting things in perspective)

$$\mathcal{L}_{\text{lr}}(\mathbf{x}, y) = \begin{cases} -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) + \log\left(1 + \exp\left(y\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)\right) & \text{if } y = +1 \text{ (positive)} \\ \log\left(1 + \exp\left(-y\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)\right) & \text{if } y = -1 \text{ (negative)} \end{cases}$$

$$\mathcal{L}_{\text{perc}}(\mathbf{x}, y) = \begin{cases} 0 & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) > 0 \\ -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) \le 0 \end{cases}$$

Main differences:

- Perceptron: gradient-based optimization
- LR: probabilistic model
- Perceptron: if the data are linearly separable, perceptron is guaranteed to converge.
- LR: likelihood can never truly be maximized with a finite w vector.