# Image analysis

## 3. Neural networks

Clément Gorin
clement.gorin@univ-paris1.fr

Sorbonne School of Economics
Masters in Development Economics

# Introduction

# Introduction

Neural networks are a family of flexible predictive statistical models composed of multiple interconnected neurons

- We introduce a fundamental structure called Multi-Layer Perceptron (Rosenblatt 1958; Rumelhart et al. 1986)
- Other structures are specialised for image analysis, language processing, generative modelling, ...
- State-of-the-art performance in problems involving with high-dimensional and unstructured data

# Introduction

Networks are called “neural” because their basic structure and functioning were loosely inspired by neuroscience

- This analogy is no longer relevant as developments follows mathematical and engineering principles
- Early implementations in the 1980’s but networks have become popular over the past decade
- Computing power, software and algorithmic innovations made networks more capable and faster to train

# Structure

A network is composed of many simple non-linear functions called neurons or units denoted by $f_u$

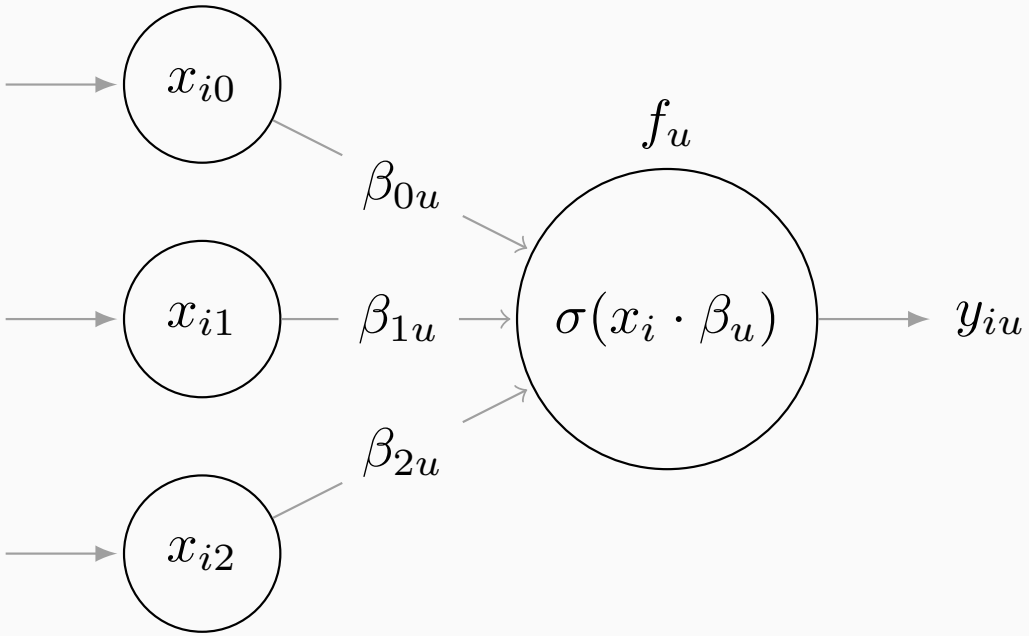
$$y_{iu} = f_u(x_i, \beta_u) = \sigma(x_i \cdot \beta_u) \tag{1}$$

where $x_i$ and $\beta_u$ are $k$-dimensional vectors $+1$ for the constant[1] and $\sigma$ is a non-linear activation function

- A unit performs a dot product between a vector of variables and parameters $\rightarrow$ linear model
- One possible form for the activation function is the sigmoid function $\rightarrow$ logistic model

[1] This is equivalent to $\sigma\left(\sum_0^k x_k \beta_k\right)$ where $x_0 = 1$.

## Network unit with two inputs and a constant



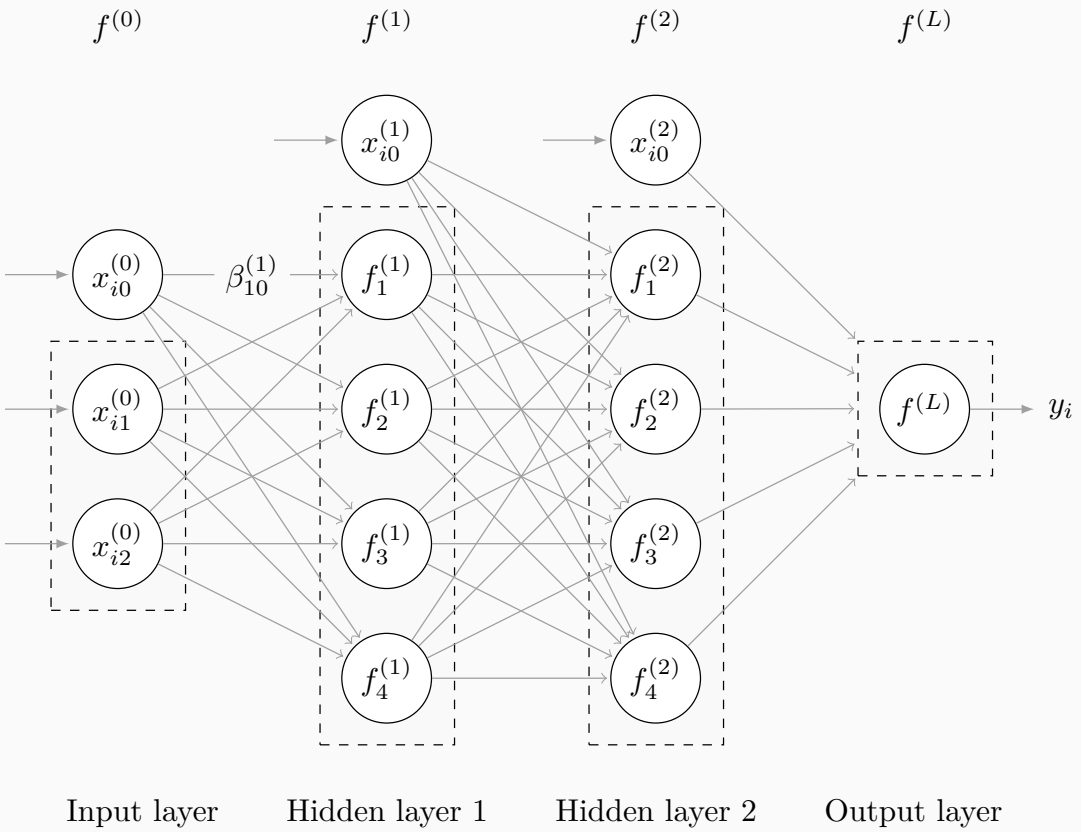

A network can be represented as a directed graph. Circles represent network units, simple arrows are parameters and filled arrows are the unit's inputs and output. The output variable $y_{iu}$ is a non-linear transformation of the input variables. For reasons that will become clear, input variables are represented as units despite the absence of computation.

# PyTorch logistic model

```
1  import torch
2  import torchinfo
3  from torch import nn
4
5  # Model class
6  class LogisticModel(nn.Module):
7      # Attributes
8      def __init__(self, nvars:int) -> None:
9          super(LogisticModel, self).__init__()
10         self.linear = nn.Linear(nvars, 1)
11         self.activation = nn.Sigmoid()
12
13     # Methods
14     def forward(self, x:torch.Tensor) -> torch.Tensor:
15         y = self.activation(self.linear(x))
16         return y
17
18 # Model instance
19 model = LogisticModel(nvars=2)
20 print(torchinfo.summary(model, input_size=(10, 2)))
21
22 #> Layer            Output Shape    Param #
23 #> ----------------------------------------
24 #> Linear (1-1)     [10, 1]         3
25 #> Sigmoid (1-2)    [10, 1]         0
```

Importantly, the unit's output is not the model's prediction but an intermediate transformation of the input

- Multiple transformations are computed using nested groups of units called layers $f^{(l)}$, $l = 0, \ldots, L$
- Hidden layers $f^{(1)}, \ldots, f^{(L-1)}$ compute recursively intermediate transformations of the input
- The output layer $f^{(L)}$ produces the model prediction in the transformed inputs i.e. the output of $f^{(L-1)}$



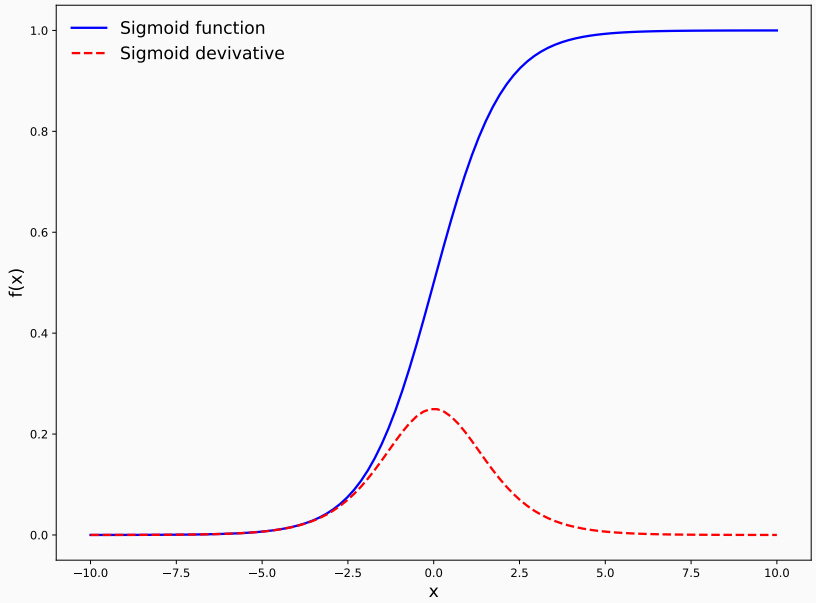

## Network with two hidden layers

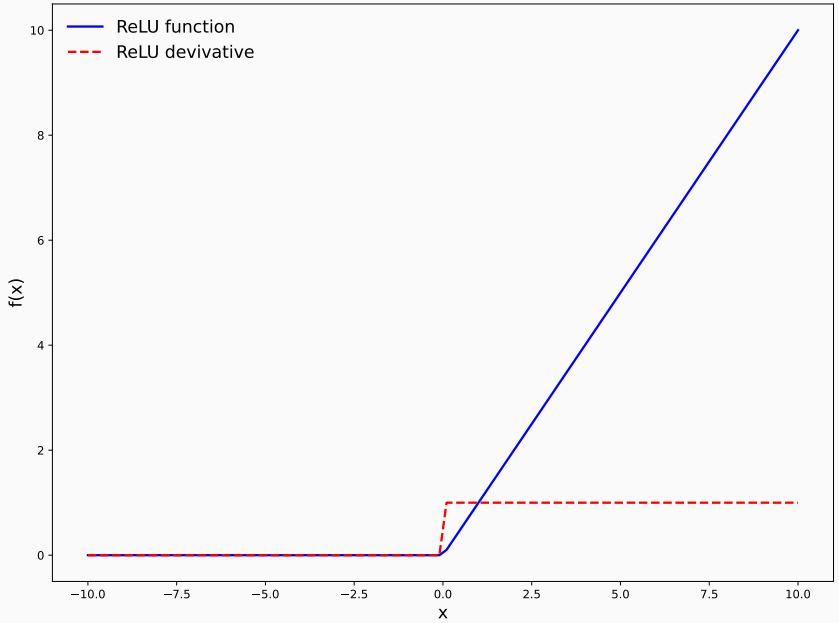
Dashed rectangles represent network layers. There model contains two hidden layers with $k^{(1)} = k^{(2)} = 4$ hidden units. Each of these $k^{(l)}$ units transform the same inputs using different parameters.

Hidden units produce intermediate transformations by computing non-linear combinations of their input

- These units can be seen as capturing the functional form from the data i.e. non-linearities and interactions
- The structure of the hidden layers – the number of layers and units they contain – are determined in advance
- Activations are chosen to help optimisation e.g. differentiable, monotonous, zero-centred, efficiency

# Common activations functions and their derivatives




Sigmoid (right) and Rectified Linear Unit (left) activation functions. With linear activation the network would simplify to a linear model: A combination of multiple linear regressions is a linear regression. Many differentiable functions can be used in the hidden layers but some have computational advantages for optimisation e.g. mitigates vanishing or exploding gradients.

Output unit(s) transform the output of the last hidden layer into the model's final prediction

- An output unit can be seen as a Generalised Linear Model (GLM) using the transformed variables as input[2]
- The number of output units and their activation function is determined by the output distribution
- Output activations include GLM (inverse) link functions e.g. identity, (multinomial) logistic, Poisson, ...

[2] Hidden units can be viewed as transforming the input variables so that the prediction problem becomes linear in the transformed space.

# PyTorch MLP (1/2)

```
# Model class
class SimpleMLP(nn.Module):

    def __init__(self, nvars:int) -> None:
        super(SimpleMLP, self).__init__()
        self.hidden_layer1 = nn.Linear(nvars, 4)
        self.hidden_layer2 = nn.Linear(4, 4)
        self.output_layer  = nn.Linear(4, 1)
        self.hidden_activ  = nn.ReLU()
        self.output_activ  = nn.Sigmoid()

    def forward(self, x:torch.Tensor) -> torch.Tensor:
        h = self.hidden_activ(self.hidden_layer1(x))
        h = self.hidden_activ(self.hidden_layer2(h))
        y = self.output_activ(self.output_layer(h))
        return y

# Model instance
model = SimpleMLP(nvars=2)
```

# PyTorch MLP (2/2)

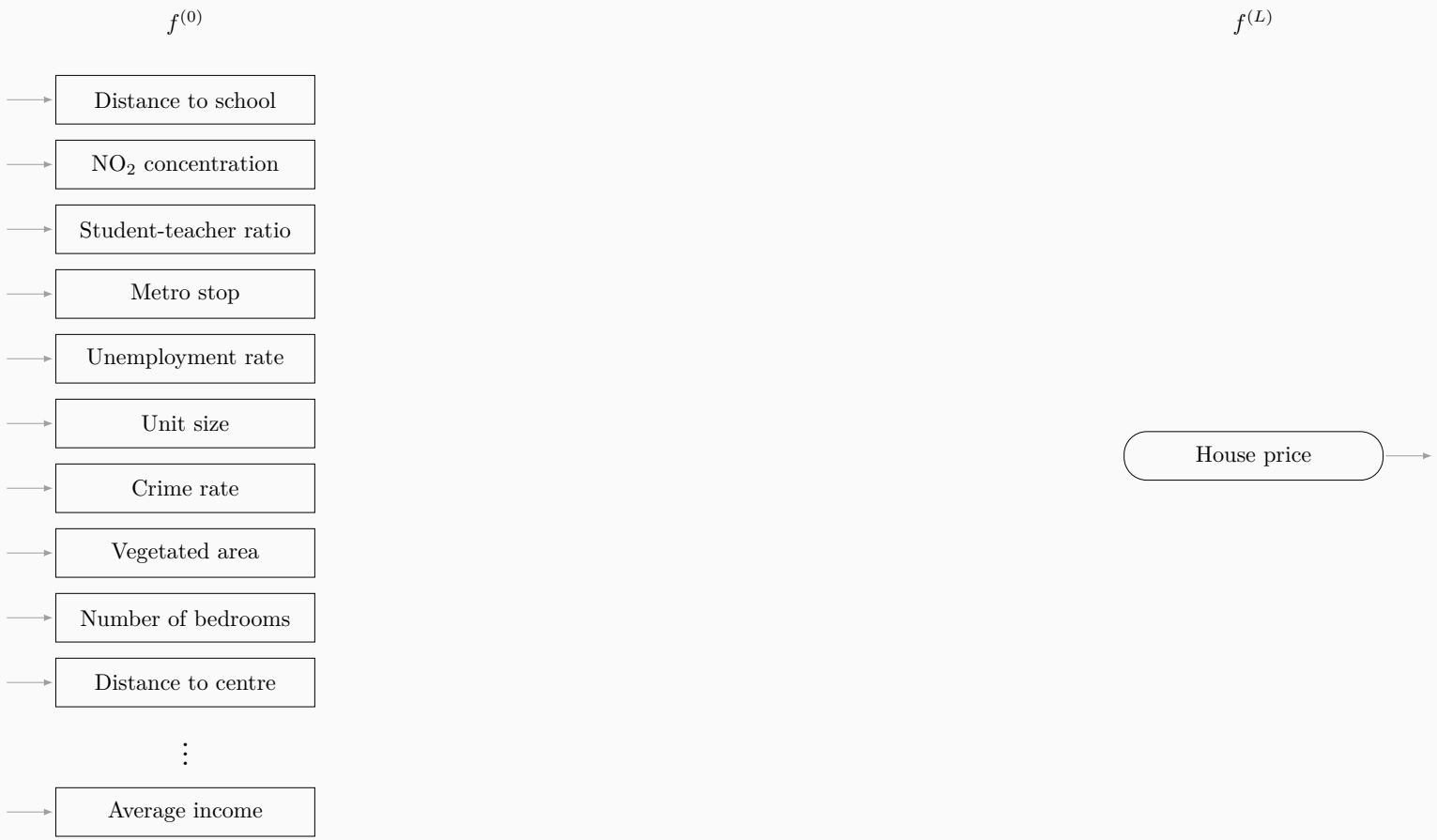
```
print(torchinfo.summary(model, input_size=(10, 2)))

#> Layer           Output Shape   Param #
#> ---------------------------------------
#> Linear (1-1)    [10, 4]        12
#> ReLU (1-2)      [10, 4]        0
#> Linear (1-3)    [10, 4]        20
#> ReLU (1-4)      [10, 4]        0
#> Linear (1-5)    [10, 1]        5
#> Sigmoid         [10, 1]        0
#> ---------------------------------------
#> Total params: 37

# Simulation
x = torch.randn(10, 2)
y = model(x)

print(y.shape)

#> torch.Size([10, 1])
```

# Representations

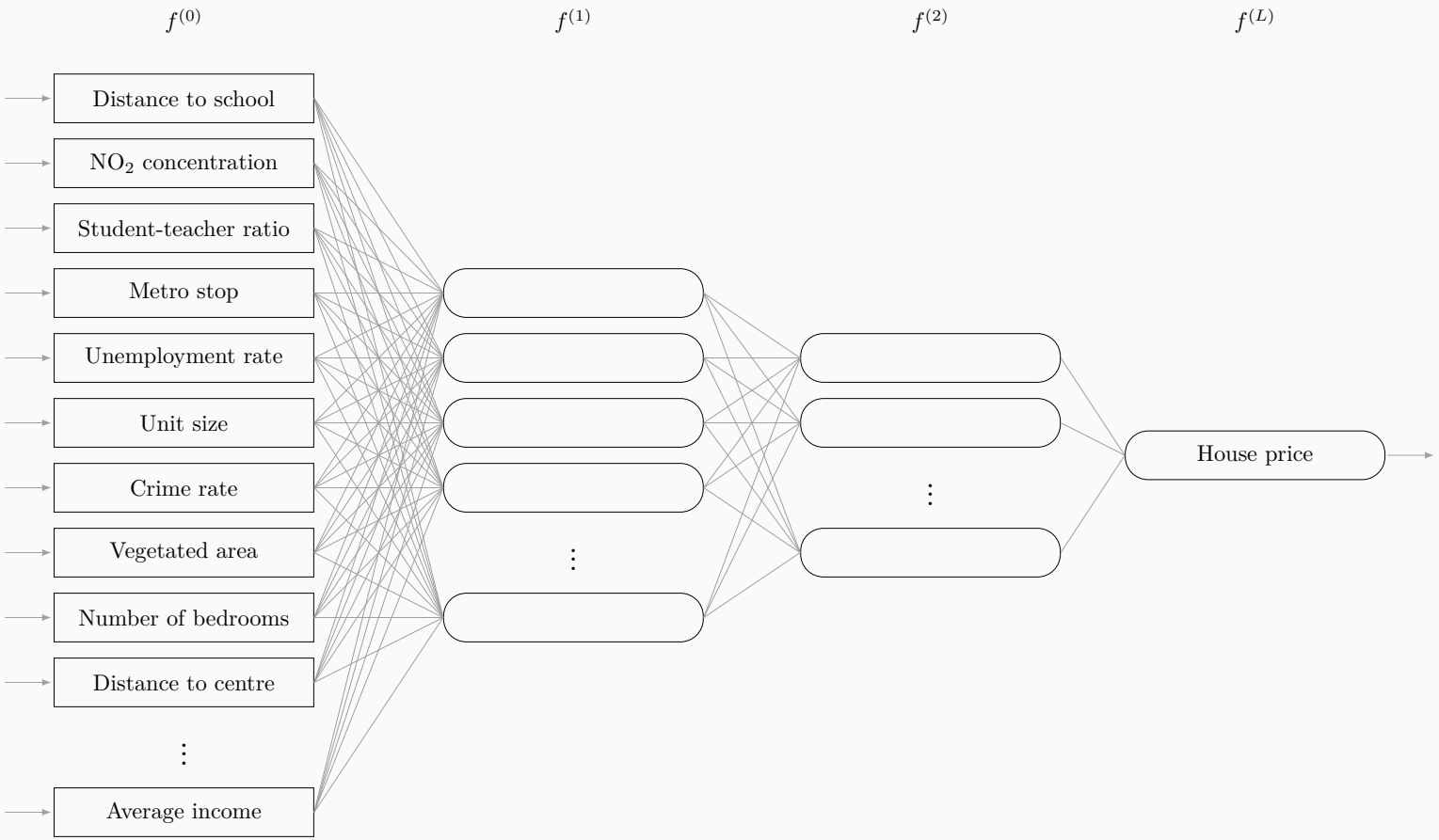
How does this distributed and hierarchical (i.e. layered) model structure increase predictive performance?

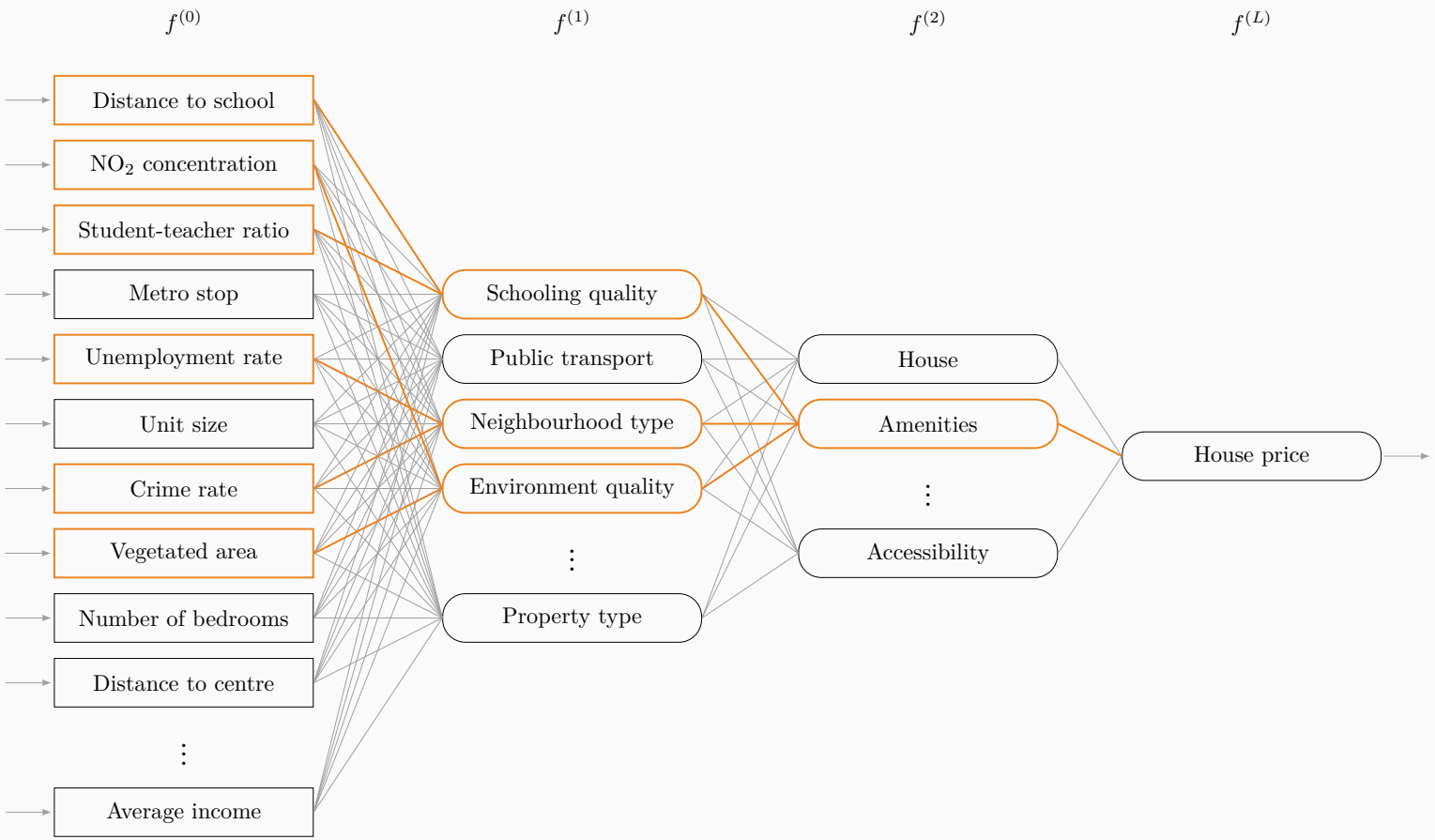
- The composition of numerous parametrised non-linear function gives the model considerable flexibility[3]
- Networks' functioning is best understood through representation learning (Bengio et al. 2013)
- Construct latent variables (i.e. features) representing the input in an abstract and informative manner

[3] Unlike linear models, networks do not attempt to fit the entire data space in a single equation, but gradually approach a functional form.




Consider predicting the price of a house from a large number of house and neighbourhood characteristics. Each input variable is an observable proxy for a more abstract latent concept, which may not be linearly related to the output.


$f^{(0)}$
$f^{(1)}$
$f^{(2)}$
$f^{(L)}$
Distance to school
$NO_2$ concentration
Student-teacher ratio
Metro stop
Unemployment rate
Unit size
Crime rate
Vegetated area
Number of bedrooms
Distance to centre
Average income
House price




Here's is an example of what an trained network might do. Hidden units are sub-models constructing increasingly abstract representations from the input. A single hidden unit may encode multiple such concepts so the representation is not directly interpretable.

Collectively optimised systems of sub-models representing the input as compositional hierarchies (LeCun et al. 2015)

- Composition within layers: units specialise in detecting distinct features from the same input
- Hierarchy across layers: units combine previously constructed features into increasingly abstract features
- Distances: constructed features are in a numerical space where the distance between them is well defined

# Generalisation

# Generalisation

Equation (1) can generalised to multiple observation and units. Note that $X^{(l)} = y^{(l)}$ due to the nested structure

$$X^{(l)} = \sigma^{(l)} \left( X^{(l-1)}_{\leftarrow x_0} \cdot \beta^{(l)} \right) \tag{2}$$

where $X^{(l)}$ and $\beta^{(l)}$ are matrices of concatenated observations (as rows) and parameters (as columns)

$$X^{(l)} = \begin{bmatrix} - & x_1^{(l)} & - \\ & \vdots & \\ - & x_n^{(l)} & - \end{bmatrix}_{n \times k^{(l)}} \qquad \beta^{(l)} = \begin{bmatrix} | & & | \\ \beta_0^{(l)} & \cdots & \beta_{k^{(l)}}^{(l)} \\ | & & | \end{bmatrix}_{k_{+1}^{(l-1)} \times k^{(l)}}$$

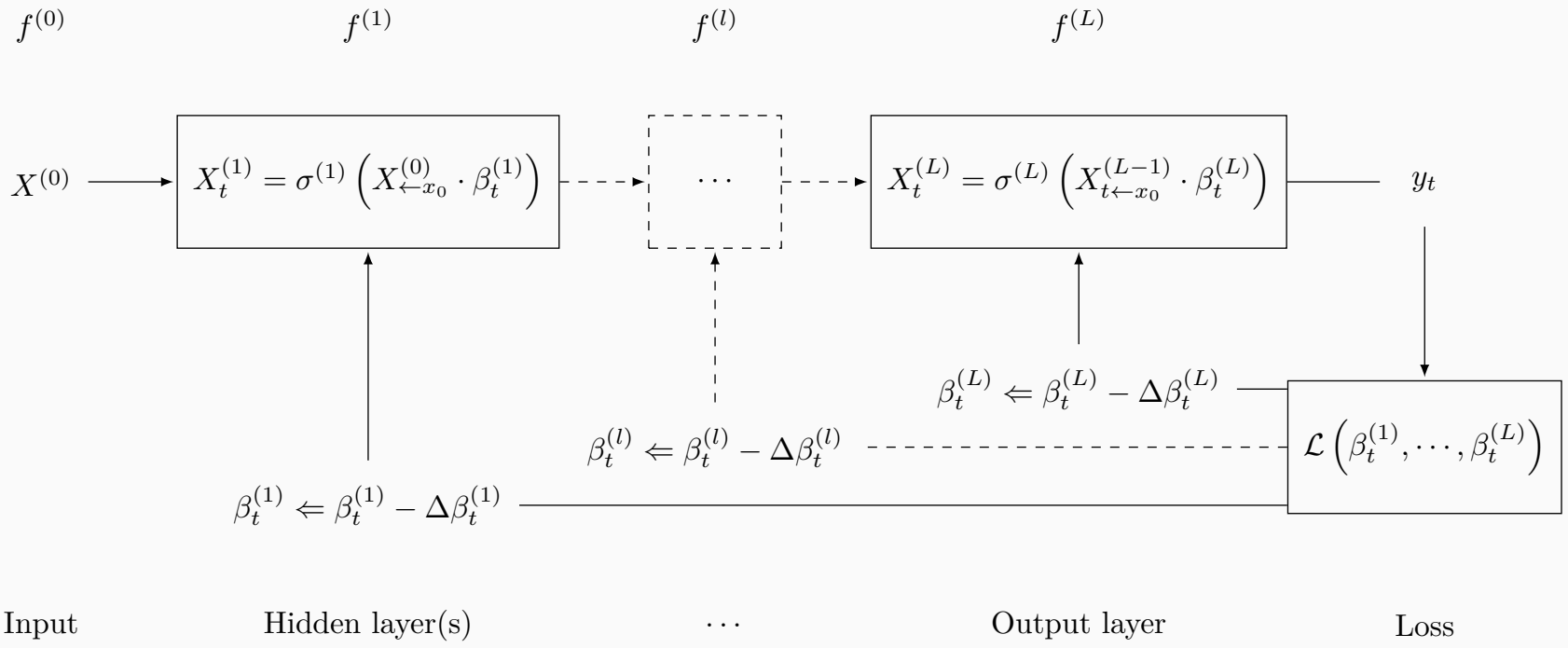
▸ Matrix product

## Forward propagation

| | Operations | | Dimensions |
|---|---|---|---|
| $f^{(1)}$ | $X^{(1)}$ | $= \sigma^{(1)}\left(X^{(0)}_{\leftarrow x_0} \cdot \beta^{(1)}\right)$ | $n \times 4 = [n \times 2_{+1}] \cdot [3 \times 4]$ |
| $f^{(2)}$ | $X^{(2)}$ | $= \sigma^{(2)}\left(X^{(1)}_{\leftarrow x_0} \cdot \beta^{(2)}\right)$ | $n \times 4 = [n \times 4_{+1}] \cdot [5 \times 4]$ |
| $f^{(L)}$ | $\hat{y} = X^{(L)}$ | $= \sigma^{(L)}\left(X^{(2)}_{\leftarrow x_0} \cdot \beta^{(L)}\right)$ | $n \times 1 = [n \times 4_{+1}] \cdot [5 \times 1]$ |

Forward propagation operations for the network illustrated in figure (2) along with the corresponding matrix dimensions. More generally, $X^{(l)}$ has dimensions $n \times k^{(l)}$ while $\beta^{(l)}$ has dimensions $k^{(l-1)}_{+1} \times k^{(l)}$.

## Abstract network




The subscript $t$ indicates the quantities that change with each iteration of the optimisation routine, $\Leftarrow$ is the update operator and $\Delta\beta_t^{(l)}$ is the amount by which the parameters are updated. The loss is computed at the end of the forward pass. The backward pass updates the parameters iteratively to decrease the training error.

# Estimation

Since hidden units have no target output, the estimated error is evaluated at the end of the forward pass

- The training error is computed using the loss function corresponding to the response distribution
- Regularisation $\rho$ may sum the parameters, either squared or expressed as an absolute value (i.e. $L_1$, $L_2$)
- There are many other ways to regularise the model (e.g. early stopping, dropout, normalisation, adversarial)

Within this parameter space, the optimisation searches the $\hat{\beta}$ that minimise the (penalised) training error

- The loss function is never strictly convex and the optimisation problem has no closed-form solution
- Gradient-based optimisation updates the parameters iteratively to converge toward a local minimum
- Gradients are computed for each training observation and averaged before updating the parameters

For simplicity consider a single training observation $i$ and a single parameter of unit $u$ indexed $j$. The update rule is

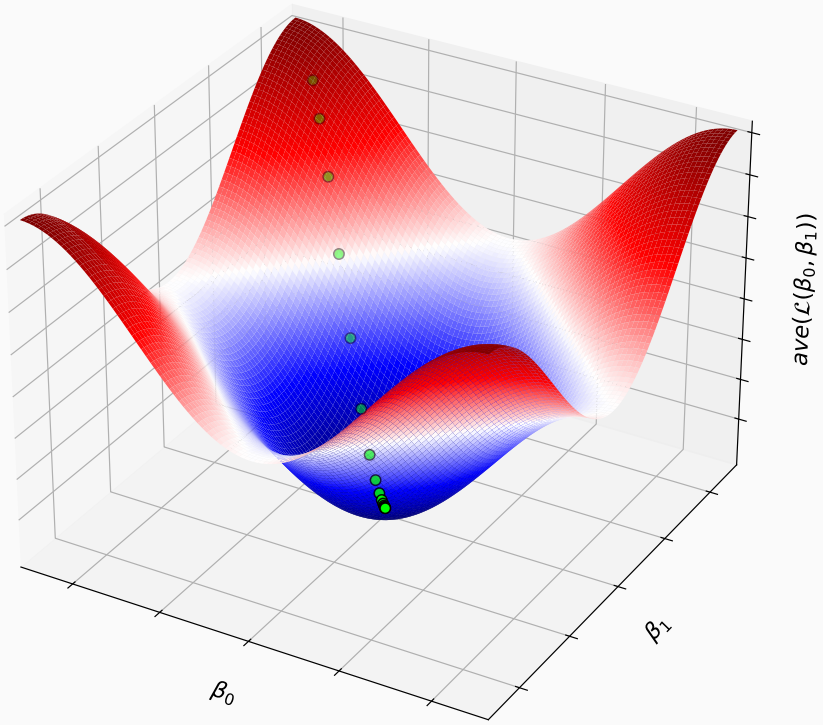
$$\beta_j \Leftarrow \beta_j - \eta \frac{\partial \mathcal{L}_i}{\partial \beta_j} \tag{3}$$

where $\Leftarrow$ is the update operator and $\eta$ is a tuning parameter called the learning rate

- A parameter is updated according to its relative contribution to the training error
- This is measured the partial derivative of the loss function with respect to this parameter

The closest local approximation of a function is the tangent, whose slope is the partial derivative

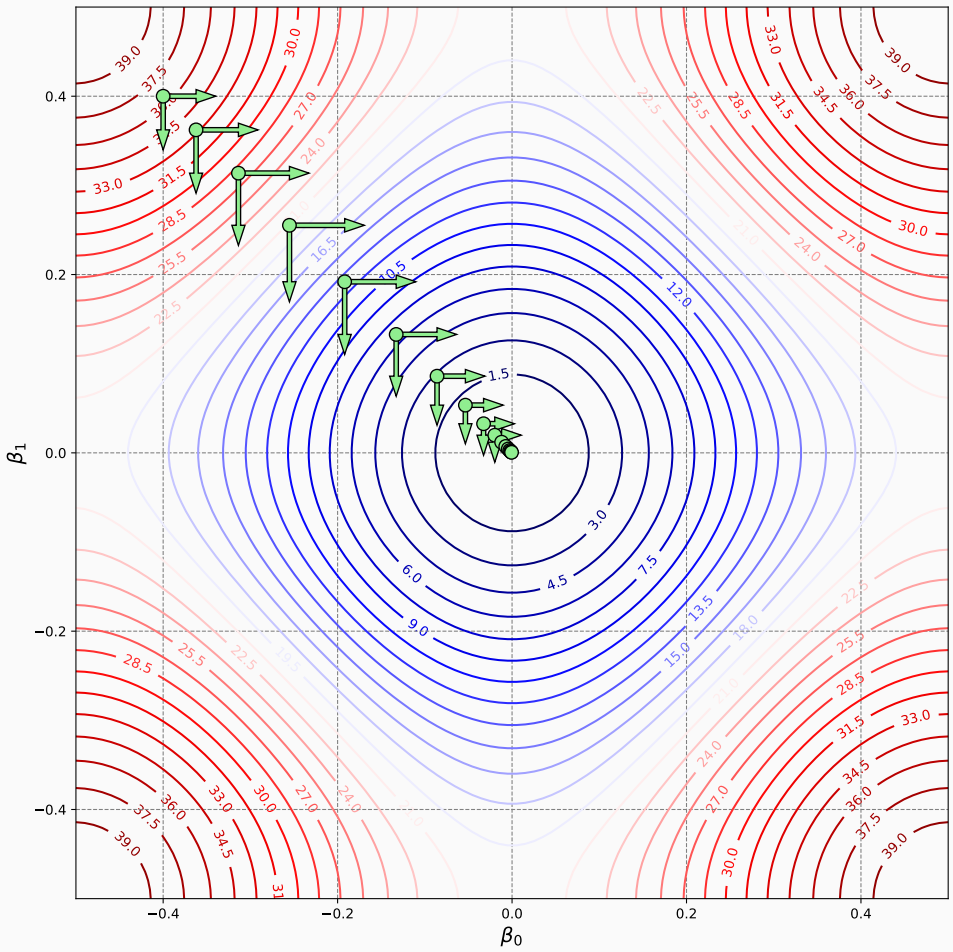
- The vector of partial derivatives with respect to every function parameter is called the gradient
- The gradient points to the direction where the function is increasing by the largest amount
- The update is proportional to the partial derivative but smaller in magnitude (i.e. local approximation)




## Loss function with two parameters

Illustrative function with two parameters, but the intuition applies to higher dimensions. The loss function averaged across training observations can be represented as a hilly landscape in the multi-dimensional space of parameter values.

▸ Loss landscape




## Loss function with two parameters

The partial derivatives of the loss function with respect to $\beta_0$ and $\beta_1$ (i.e. the gradient vector) indicate the direction in which the loss function increases most rapidly. The parameters are updated in the direction of the negative gradient multiplied by the learning rate (i.e. green arrows).

Generalising to multiple training observations, $n$ partial derivatives are computed for each parameter

- A change in a single parameter affects the training error for every observation, either positively or negatively
- Partial derivatives are averaged out to compute the update that reduces the average training error
- The size of the training data and the number of parameters affect the computation cost of the update

## Pytorch training

```
from torch import optim, utils
from tqdm import tqdm

# Defines device
device = (
    'cuda' if torch.cuda.is_available() else # NVIDIA
    'mps'  if torch.backends.mps.is_available() else # Apple Silicon
    'cpu'
)
device = torch.device(device)

# Loss function
criterion = nn.MSELoss(reduction='mean')

# Optimiser
optimiser = optim.AdamW(model.parameters(), lr=1e-3)

# Data loader
train_dataset = utils.data.TensorDataset(torch.randn(1000, 2), torch.randn(1000, 1))
train_loader  = utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
```

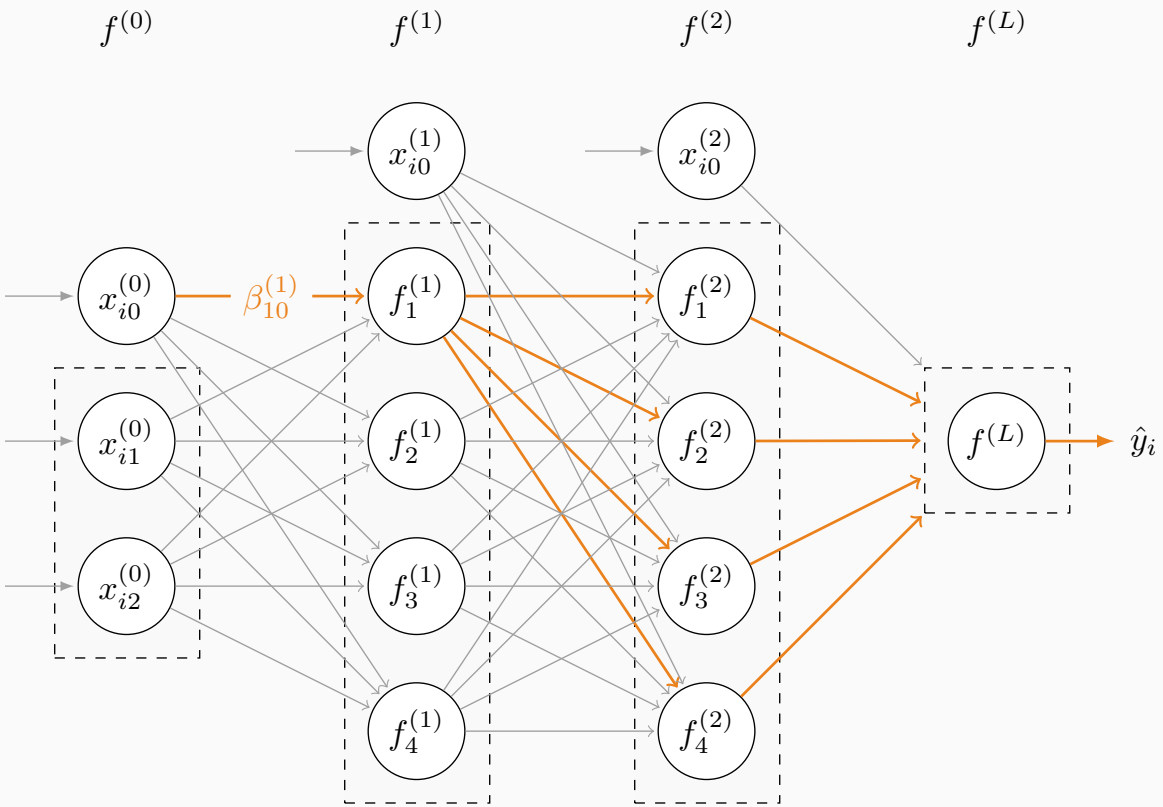
## Pytorch training

```
# Model training
model.to(device) # Moves the model to the device
model.train()    # Sets the model to training mode
n_epochs = 10    # Number of epochs

for epoch in range(n_epochs):
    tot_obs, tot_loss = 0, 0
    for x, y in tqdm(train_loader, desc=f'Epoch {epoch+1}/{n_epochs}', unit='batch'):
        x, y = x.to(device), y.to(device) # Moves the data to the device
        # Training step
        optimiser.zero_grad()   # Clears the gradients - no accumulation
        yh   = model(x)         # Forward pass
        loss = criterion(yh, y) # Computes the loss
        loss.backward()         # Backward pass - computes the gradients
        optimiser.step()        # Updates the parameters
        # Statistics
        tot_obs  += x.size(0)
        tot_loss += loss.item() * x.size(0)
    print(f'Epoch {epoch+1}/{n_epochs} - loss: {tot_loss/tot_obs:.4f}')
```

# Backpropagation

Computing the gradients is demanding given the number of parameters and the nested nature of the model

- Backpropagation (Rumelhart et al. 1986) addresses these issues using the chain rule of differentiation
- The complex gradient expressions can be expressed as the product of simple partial derivatives
- Duplicate calculations are avoided by using quantities that are computed during the forward pass




## Backpropagation

A small change to $\beta_{10}^{(1)}$ alters the result of the dot product $z_1^{(1)}$ and the unit's output $x_1^{(1)}$. This modifies the output of the four units in the second hidden layer, the predicted response and training error. To compute the contribution of a parameter, we take into account its impact on all units in the next layers

## Chain rule

Consider the function composition

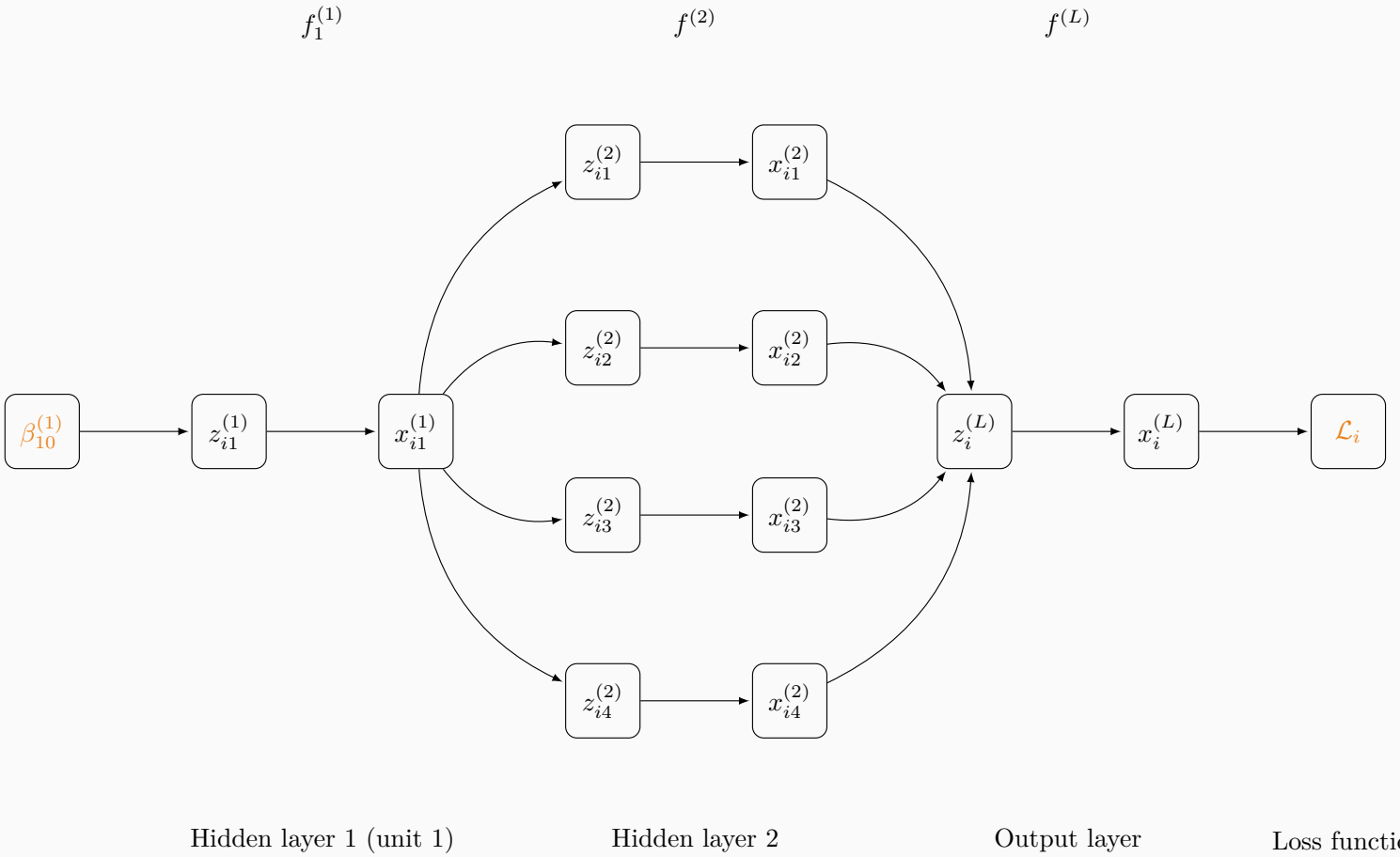
$$y = f(x)$$
$$z = g(y) = (g \circ f)(x)$$

The chain rule states that

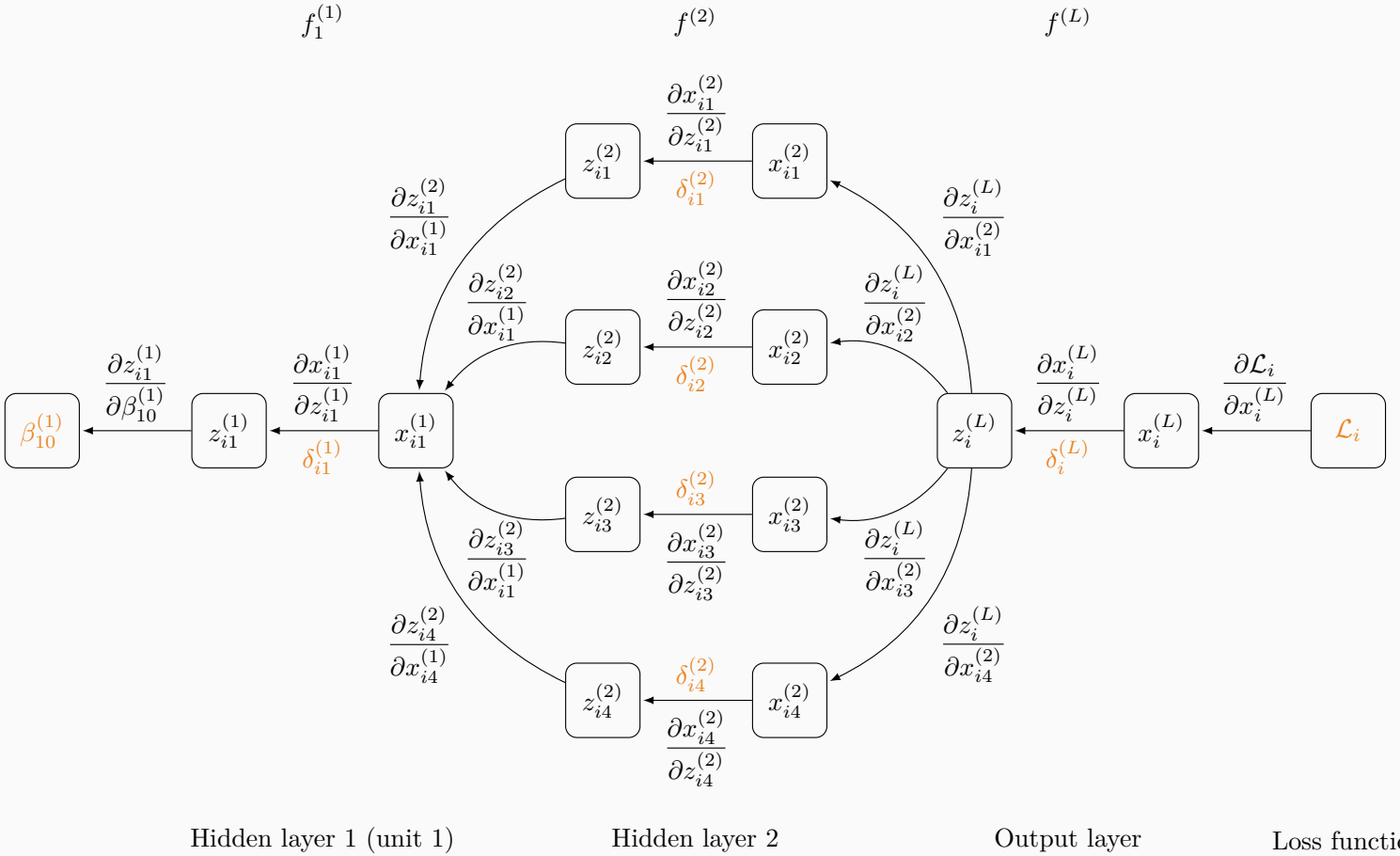
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x}$$

The complex gradient expressions can be expressed as the product of simple partial derivatives

# Impact of changing $\beta_{10}^{(1)}$ on network



# Detailed backpropagation operations



The partial derivative of the loss with respect to $\beta_{01}^{(1)}$ for observation $i$ can be expressed as

$$\frac{\partial \mathcal{L}_i}{\partial \beta_{10}^{(l)}} = \underbrace{\frac{\partial \mathcal{L}_i}{\partial x_i^{(L)}} \frac{\partial x_i^{(L)}}{\partial z_i^{(L)}}}_{\delta_i^{(L)}} \underbrace{\frac{\partial z_i^{(L)}}{\partial x_{i1}^{(2)}} \frac{\partial x_{i1}^{(2)}}{\partial z_{i1}^{(2)}}}_{\prod \cdots = \delta_{i1}^{(2)}} \underbrace{\frac{\partial z_i^{(L)}}{\partial x_{i2}^{(2)}} \frac{\partial x_{i2}^{(2)}}{\partial z_{i2}^{(2)}}}_{\prod \cdots = \delta_{i2}^{(2)}} \underbrace{\frac{\partial z_i^{(L)}}{\partial x_{i3}^{(2)}} \frac{\partial x_{i3}^{(2)}}{\partial z_{i3}^{(2)}}}_{\prod \cdots = \delta_{i3}^{(2)}} \underbrace{\frac{\partial z_i^{(L)}}{\partial x_{i4}^{(2)}} \frac{\partial x_{i4}^{(2)}}{\partial z_{i4}^{(2)}}}_{\prod \cdots = \delta_{i4}^{(2)}} \underbrace{\frac{\partial z_{i1}^{(2)}}{\partial x_{i1}^{(1)}} \frac{\partial z_{i2}^{(2)}}{\partial x_{i1}^{(1)}} \frac{\partial z_{i3}^{(2)}}{\partial x_{i1}^{(1)}} \frac{\partial z_{i4}^{(2)}}{\partial x_{i1}^{(1)}} \frac{\partial x_{i1}^{(1)}}{\partial z_{i1}^{(1)}}}_{\ldots \delta_{i1}^{(1)}} \frac{\partial z_{i1}^{(1)}}{\partial \beta_{10}^{(1)}} \tag{4}$$

where $\prod \ldots$ denotes the product of the bracket with the lines above and $\delta_{iu}^{(l)}$ is the "error signal" defined as $\partial \mathcal{L}_i / \partial z_{iu}^{(l)}$

More generally, the partial derivative of the loss with respect to $\beta_{ju}^{(l)}$ for observation $i$ can be expressed as

$$\frac{\partial \mathcal{L}_i}{\partial \beta_{ju}^{(l)}} = \frac{\partial \mathcal{L}_i}{\partial z_{iu}^{(l)}} \frac{\partial z_{iu}^{(l)}}{\partial \beta_{ju}^{(l)}} = \delta_{iu}^{(l)} \frac{\partial z_{iu}^{(l)}}{\partial \beta_{ju}^{(l)}} \tag{5}$$

where the "error signal" $\delta_{iu}^{(l)}$ measures the contribution of $z_{iu}^{(l)}$ to the training error

- Backpropagation computes efficiently the error signals $\delta_{iu}^{(l)}$ for each unit and observation
- The error signals can be easily related to the partial derivative with respect to each parameter

To computes the error signals efficiently, backpropagation avoids duplicated computations

- The error signal of a layer $l$ can be expressed as a function of the error signal in next layer $l + 1$
- These error signals are computed starting from the output layer all the way to the first hidden layer
- For increased efficiency, the relevant calculations can be stored in memory during the forward pass

## Forward and backward passes

| | Forward propagation | | | Backpropagation | | |
|---|---|---|---|---|---|---|
| | $\downarrow$ | $X^{(0)}$ | $= X^{(0)}_{\leftarrow x_0}$ | $\leftarrow$ | | |
| $f^{(1)}$ | $\downarrow$ | $Z^{(1)}$ | $= X^{(0)} \cdot \beta^{(1)}$ | $\uparrow$ | $\partial Z^{(1)}/\partial \beta^{(1)}$ | $= X^{(0)T}$ |
| | $\downarrow$ | $X^{(1)}$ | $= \sigma(Z^{(1)})$ | $\delta(1)$ | $\partial X^{(1)}/\partial Z^{(1)}$ | $= \sigma'(Z^{(1)})$ |
| | $\downarrow$ | $X^{(1)}$ | $= X^{(1)}_{\leftarrow x_0}$ | $\uparrow$ | $\partial Z^{(2)}/\partial X^{(1)}$ | $= \beta^{(2)T}$ |
| | $\vdots$ | | | $\vdots$ | | |
| $f^{(l)}$ | $\downarrow$ | $Z^{(l)}$ | $= X^{(l-1)} . \beta^{(l)}$ | $\uparrow$ | $\partial Z^{(l)}/\partial \beta^{(l)}$ | $= X^{(l-1)T}$ |
| | $\downarrow$ | $X^{(l)}$ | $= \sigma(Z^{(l)})$ | $\delta^{(l)}$ | $\partial X^{(l)}/\partial Z^{(l)}$ | $= \sigma'(Z^{(l)})$ |
| | $\downarrow$ | $X^{(l)}$ | $= X^{(l)}_{\leftarrow x_0}$ | $\uparrow$ | $\partial Z^{(l+1)}/\partial X^{(l)}$ | $= \beta^{(l+1)T}$ |
| | $\vdots$ | | | $\vdots$ | | |
| $f^{(L)}$ | $\downarrow$ | $Z^{(L)}$ | $= X^{(L-1)} . \beta^{(L)}$ | $\uparrow$ | $\partial Z^{(L)}/\partial \beta^{(L)}$ | $= X^{(L-1)T}$ |
| | $\downarrow$ | $X^{(L)}$ | $= \sigma(Z^{(L)})$ | $\delta^{(L)}$ | $\partial X^{(L)}/\partial Z^{(L)}$ | $= \sigma'(Z^{(L)})$ |
| | $\rightarrow$ | | | $\uparrow$ | $\partial \mathcal{L}(\beta)/\partial X^{(L)}$ | $= \mathcal{L}'(X^{(L)})$ |

Most of the quantities required to compute the gradients, such as $X^{(l)}$, $\beta^{(l)}$ and $Z^{(l)}$ are stored during forward propagation to avoid duplicated computations

The matrix $\delta^{(l)}$ has dimensions $n \times k^{(l)}$ and contains the error signals for every unit and observations

$$\begin{aligned}
\delta^{(L)} &= \frac{\partial \mathcal{L}(\beta)}{\partial X^{(L)}} \frac{\partial X^{(L)}}{\partial Z^{(L)}} \\
&= \mathcal{L}'(X^{(L)}) \odot \sigma'^{(L)}(Z^{(L)}) \qquad (6) \\
\delta^{(l)} &= \delta^{(l+1)} \frac{\partial Z^{(l+1)}}{\partial X^{(l)}} \frac{\partial X^{(l)}}{\partial Z^{(l)}} \\
&= \delta^{(l+1)} \cdot \beta^{(l+1)T} \odot \sigma'^{(l)}(Z^{(l)}_{\leftarrow x_0}) \qquad (7)
\end{aligned}$$

where $\odot$ denotes the Hadamard product and the gradient is represented as a column vector (i.e. denominator layout)

Using the expression for the error signal, the matrix of partial derivatives with respect to each parameter is computed as

$$\frac{\partial \mathcal{L}(\beta)}{\partial \beta^{(l)}} = X_{\leftarrow x_0}^{(l-1)T} \cdot \delta_{\rightarrow x_0}^{(l)} \tag{8}$$

For each parameter, this formulation sums the gradients across training observations. The update rule becomes

$$\beta^{(l)} \Leftarrow \beta^{(l)} - \frac{\eta}{n} \frac{\partial \mathcal{L}(\beta)}{\partial \beta^{(l)}} \tag{9}$$

In practice, each gradient descent iteration is performed using a partition of the training observations called “batch”

| | | Operations | Dimensions |
|---|---|---|---|
| $f^{(L)}$ | $\delta^{(L)}$ | $= \mathcal{L}'(X^{(L)}) \odot \sigma^{(L)'}(Z^{(L)})$ | $n \times 1 = [n \times 1] \odot [n \times 1]$ |
| | $\frac{\partial \mathcal{L}(\beta)}{\partial \beta^{(L)}}$ | $= X^{(2)T}_{\leftarrow x_0} \cdot \delta^{(L)}$ | $5 \times 1 = [4_{+1} \times n] \cdot [n \times 1]$ |
| $f^{(2)}$ | $\delta^{(2)}$ | $= \delta^{(L)} \cdot \beta^{(L)T} \odot \sigma^{(2)'}(Z^{(2)}_{\leftarrow x_0})$ | $n \times 5 = [n \times 1] \cdot [1 \times 5] \odot [n \times 4_{+1}]$ |
| | $\frac{\partial \mathcal{L}(\beta)}{\partial \beta^{(2)}}$ | $= X^{(1)T}_{\leftarrow x_0} \cdot \delta^{(2)}_{\rightarrow x_0}$ | $5 \times 4 = [4_{+1} \times n] \cdot [n \times 5_{-1}]$ |
| $f^{(1)}$ | $\delta^{(1)}$ | $= \delta^{(2)}_{\rightarrow x_0} \cdot \beta^{(2)T} \odot \sigma^{(1)'}(Z^{(1)}_{\leftarrow x_0})$ | $n \times 5 = [n \times 5_{-1}] \cdot [4 \times 5] \odot [n \times 4_{+1}]$ |
| | $\frac{\partial \mathcal{L}(\beta)}{\partial \beta^{(1)}}$ | $= X^{(0)T}_{\leftarrow x_0} \cdot \delta^{(1)}_{\rightarrow x_0}$ | $3 \times 4 = [2_{+1} \times n] \odot [n \times 5_{-1}]$ |

The denominator layout implies the inversion of the two terms and the transpose for the matrix multiplications. Note that $\delta^{(l+1)} \cdot \beta^{(l+1)T}$ contains the bias while $\sigma'(Z^{(l)})$ does not. To allow for element-wise multiplication, a column of ones is added to $Z^{(l)}$. When we back-propagate to the previous layer, $\delta^{(l)}$ contains the error linked to the bias, which is removed because it is not connected to the previous layer.

▸ Gradient checking

# Interpretation

# Interpretation

There are methods to explain the response by quantifying how much each input feature contributes to the prediction[4]

- Local (i.e. observation) and global (i.e. sample) importance statistics are not readily available for networks
- **Gradient-based methods** e.g. Integrated Gradients
  $\rightarrow$ specific to networks, highly efficient
- **Perturbation-based methods** e.g. Ablation (Zeiler and Fergus 2014), SHAP (Lundberg and Lee 2017)

[4] Perturbation-based methods are model agnostic. We will cover other methods for specific models structures e.g. saliency maps (CNN), attention visualisation (transformers).

Integrated Gradients (Sundararajan et al. 2017) computes the contribution of each input feature to a model's prediction

$$IG(x_i) = (x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial f(x^\alpha)}{\partial x_i} \, d\alpha \tag{10}$$

$$x^\alpha = x' + \alpha \cdot (x - x')$$

where $x_i$ is the actual value, $x_0$ is the baseline value (e.g. $0$ or sample mean) and $\alpha \in [0, 1]$ a scaling factor

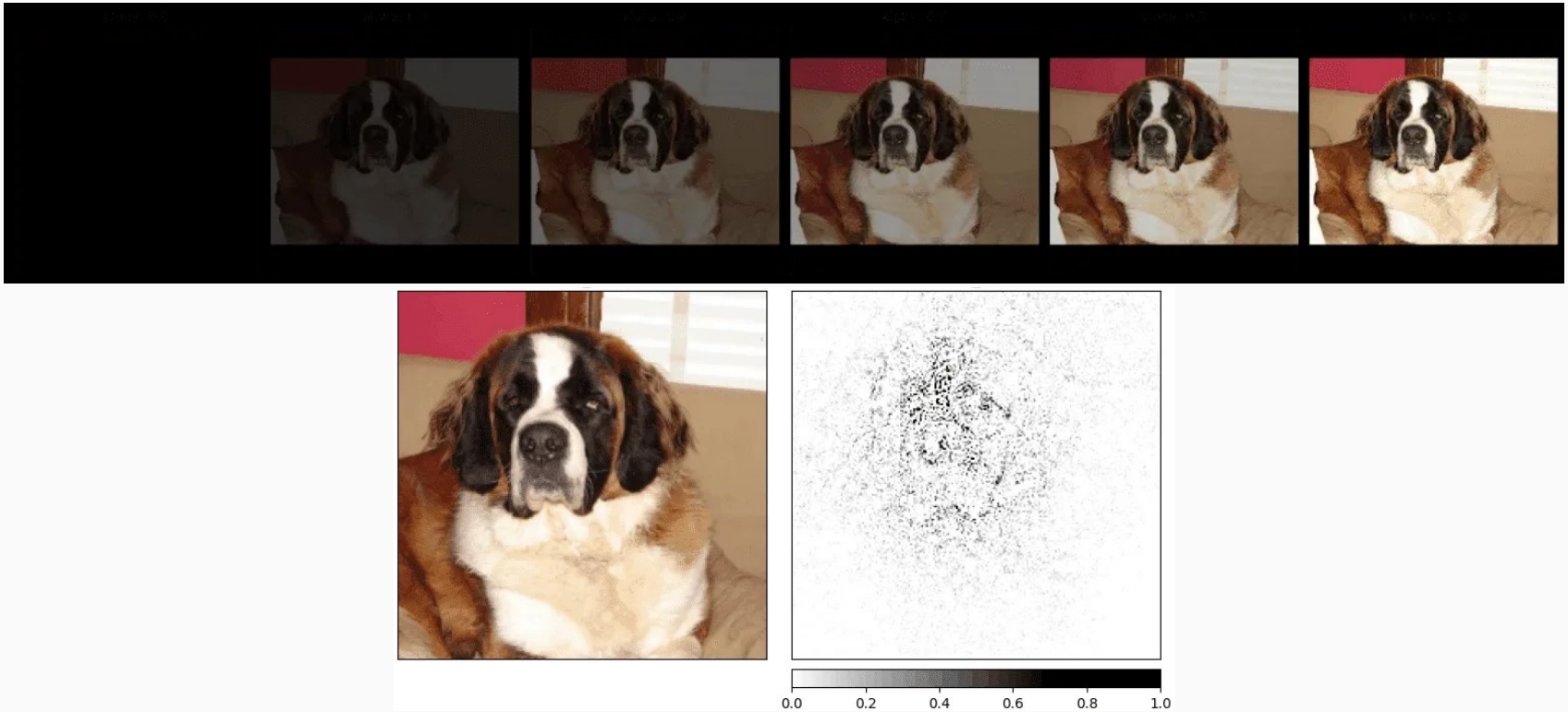
- This captures the contribution of $x_i$ to changes in $y_i$ as $x_i$ moves from the baseline to its actual value
- The individual IGs can be averaged across the entire test sample to compute global feature importance

▸ Numerical implementation

## Feature importance

```
from captum import attr

# Input and baseline
model.eval().to('cpu')
inputs   = torch.randn(100, 2, dtype=torch.float32, requires_grad=True).to(device)
baseline = torch.zeros_like(inputs, dtype=torch.float32).to(device)

# Integrated Gradients
explain_IG = attr.IntegratedGradients(model)
attribs_IG = explain_IG.attribute(inputs, baselines=baseline, target=0)
attribs_IG = attribs_IG.detach().numpy()

# Feature ablation
explain_FA = attr.FeatureAblation(model)
attribs_FA = explain_FA.attribute(inputs)
attribs_FA = attribs_FA.detach().numpy()

# SHAP values
explain_GS = attr.GradientShap(model)
attribs_GS = explain_GS.attribute(inputs=inputs, baselines=baseline)
attribs_GS = attribs_GS.detach().numpy()
```

## Integrated Gradients example




**Top:** Extrapolation from baseline (0 or black) to actual RGB values. **Bottom:** Integrated gradients for the input pixels, summed across colour channels. The scores measure the relative contribution of input pixels to the output. Source: Kemal Erdem.

# Summary

# Summary

The processing of high-dimensional and unstructured data offer strong potential for original economic analysis

- Traditional models perform poorly due to the dimensionality and the absence of theory
- Neural networks are powerful and flexible predictive models designed to solve these problem
- Specific networks offer state-of-the art performance on image an text data among others

# Summary

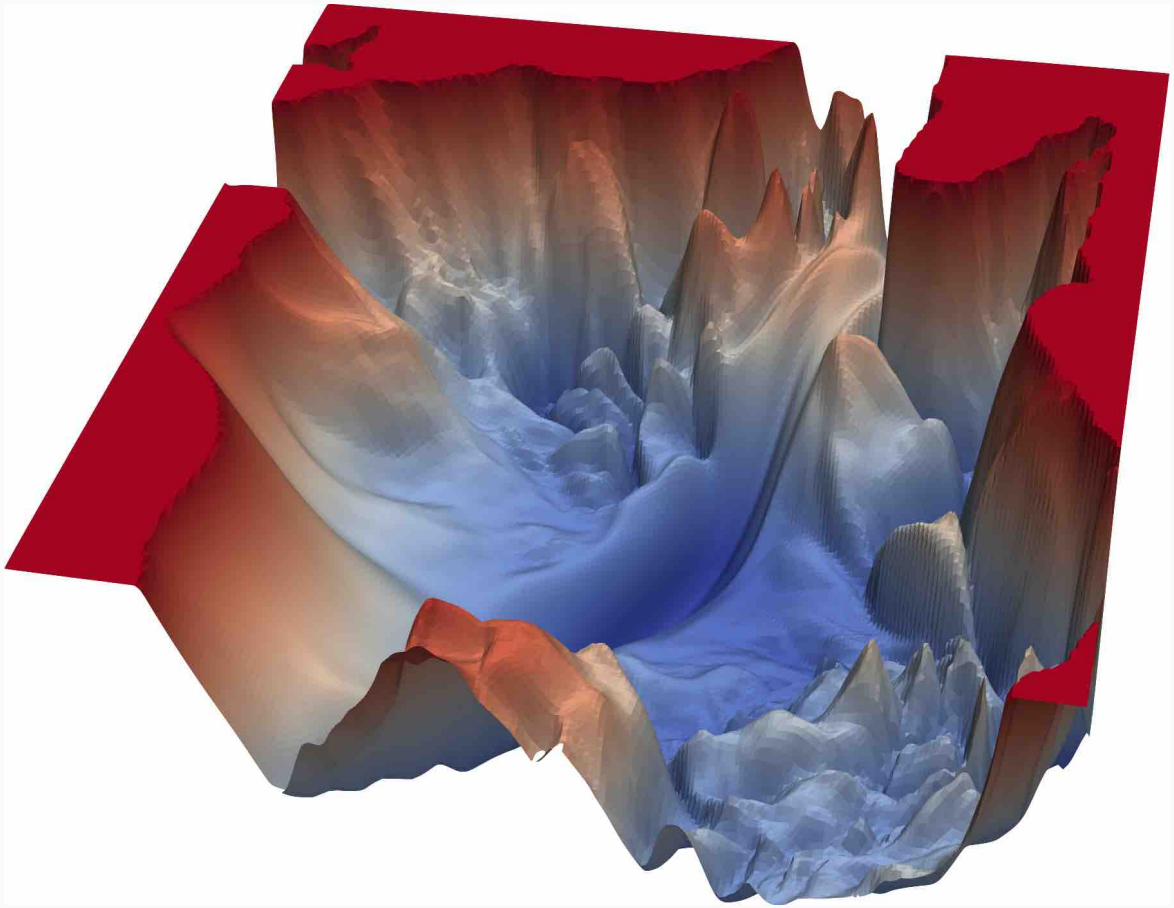
On the downside networks trade most interpretability for flexibility and are computationally heavy

- Approximating even simple functions involve many parameters and requires more observations
- Many tuning parameters e.g. layers, units, activation, loss, optimiser, regularisation, learning rate
- Network structures and implementation details and are important in this more empirical literature

# Summary

Computing the gradients is demanding given the number of parameters and the nested nature of the model

- The backpropagation algorithm addresses these issues by using the chain rule of differentiation
- The complex gradient expressions can be expressed as a product of simple partial derivatives
- It avoids duplicate calculations by re-using quantities that are computed and stored during the forward pass



## Illustrative loss landscape

Three dimensional representation of a VGG-56 network loss landscape (Li et al. 2018). Complex loss landscapes have many local minima and saddle points where the optimisation could get trapped and converge to sub-optimal values.

Thank you for your attention!

# References

# References i


- Rosenblatt, Frank (1958). **“The perceptron: A probabilistic model for information storage and organization in the brain”.** In: *Psychological Review* 65.6, pp. 386–408 (cit. on p. 3).
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). **“Learning representations by back-propagating errors”.** In: *Nature* 323.6088, pp. 533–536 (cit. on pp. 3, 38).
- Bengio, Yoshua, Aaron Courville, and Pascal Vincent (2013). **“Representation learning: A review and new perspectives”.** In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.8, pp. 1798–1828 (cit. on p. 18).

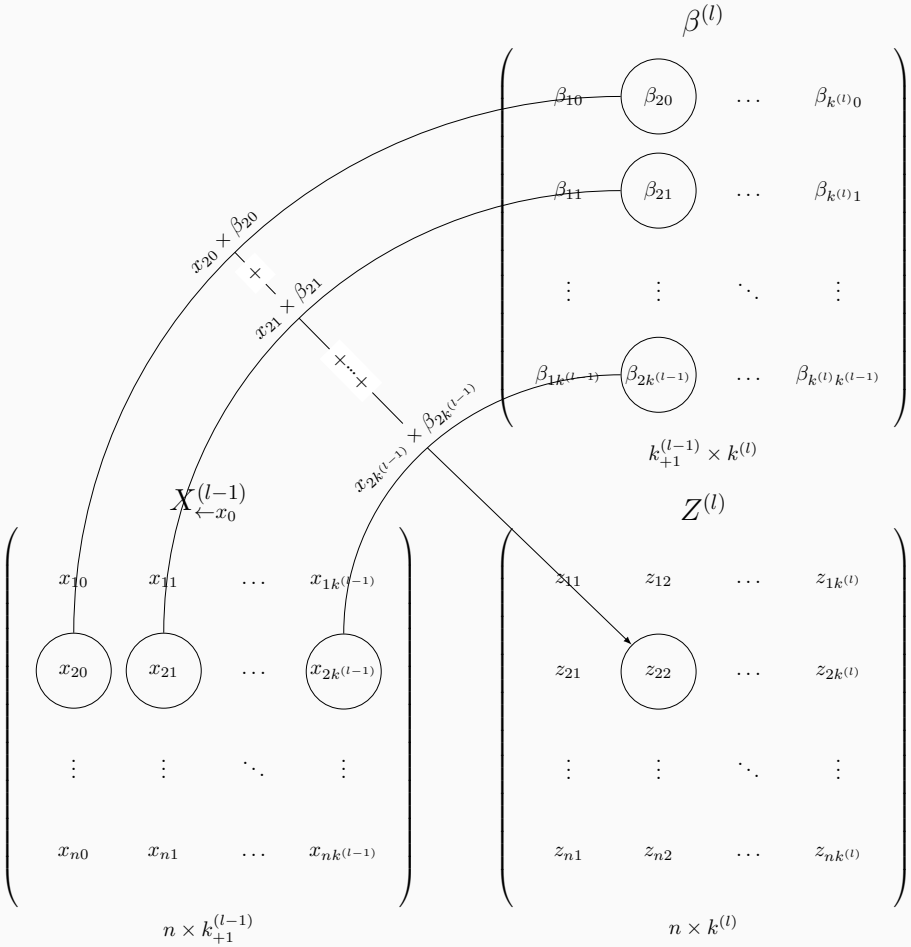
# References ii


- Zeiler, Matthew D. and Rob Fergus (2014). **"Visualizing and understanding convolutional networks".** In: *Computer Vision - ECCV 2014*, pp. 818–833 (cit. on p. 51).
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). **"Deep learning".** In: *Nature* 521, pp. 436–444 (cit. on p. 22).
- Nielsen, Michael A. (2015). ***Neural networks and deep learning*.** Determination Press.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). ***Deep Learning*.** MIT Press.
- Lundberg, Scott M and Su-In Lee (2017). **"A Unified Approach to Interpreting Model Predictions".** In: *Advances in Neural Information Processing Systems*. Vol. 30 (cit. on p. 51).

# References iii


- Sundararajan, Mukund, Ankur Taly, and Qiqi Yan (2017). **“Axiomatic Attribution for Deep Networks”.** In: *CoRR* abs/1703.01365 (cit. on p. 52).
- Li, Hao et al. (2018). **“Visualizing the loss landscape of neural nets”.** In: *Advances in Neural Information Processing Systems*. Vol. 31 (cit. on p. 59).

# Appendix

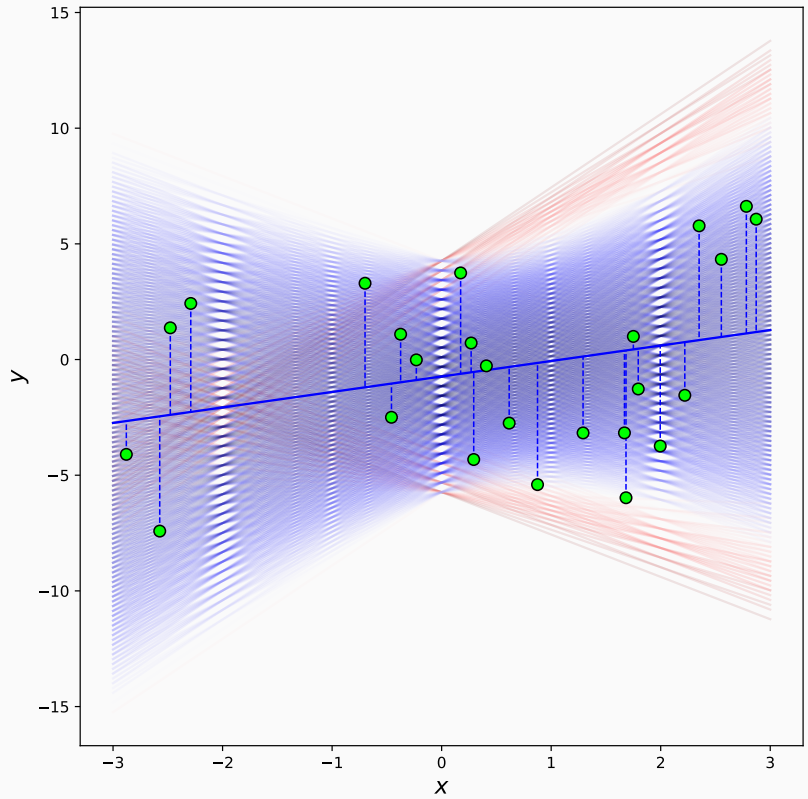



## Matrix product

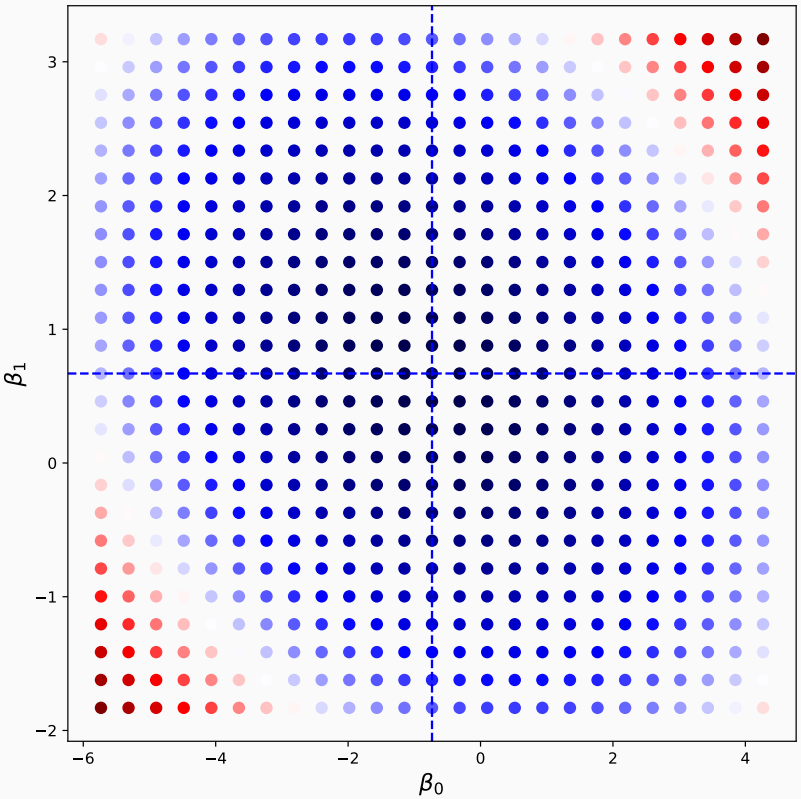
$X_{\leftarrow x_0}^{(l-1)}$ contains $n$ observations (rows) and $k_{+1}^{(l-1)}$ variables (columns), including the constant. $\beta^{(l)}$ contains the corresponding $k_{+1}^{(l-1)}$ parameters for each of the $k^{(l)}$ units (columns). The product of these two matrices is denoted $Z^{(l)}$ with $n$ observations, where each $k^{(l)}$ variable represents a combination of the same input with different parameters. The activation function $\sigma^{(l)}$ is applied element-wise to produce the matrix $X^{(l)}$

◂ Presentation

## Parameter combinations




## Loss landscape




◂ Presentation

You can check your implementation of backpropagation against a numerical approximation of the gradient

$$\frac{\partial \mathcal{L}(\beta)}{\partial \beta} \approx \frac{\mathcal{L}(\beta + \epsilon) - \mathcal{L}(\beta - \epsilon)}{2\varepsilon}$$

where $\varepsilon$ is a small random value. In the case where $\beta$ is a vector of values, we can compute for each $\beta_j,\ j = 1, \ldots, k$

$$\frac{\partial \mathcal{L}(\beta_)}{\partial \beta_0} \approx \frac{\mathcal{L}(\beta_0 + \epsilon, \beta_1, \ldots, \beta_k) - \mathcal{L}(\beta_0 - \epsilon, \beta_1, \ldots, \beta_k)}{2\varepsilon}$$
$$\frac{\partial \mathcal{L}(\beta_)}{\partial \beta_1} \approx \frac{\mathcal{L}(\beta_0, \beta_1 + \epsilon, \ldots, \beta_k) - \mathcal{L}(\beta_0, \beta_1 - \epsilon, \ldots, \beta_k)}{2\varepsilon}$$
$$\ldots$$

In practice, we use a discrete approximation because computing the integral is computationally infeasible

$$IG(x_i) \approx (x_i - x_i^0) \sum_{k=1}^{K} \frac{1}{K} \frac{\partial f(x^{\alpha_k})}{\partial x_i} \tag{11}$$

where $K$ is the number of steps e.g. $K = 50$

- The approximation uses the trapezoidal rule i.e. assumes approximately linear gradients across consecutive steps
- The gradients at each pair of consecutive points are averaged and scaled by $\frac{1}{2}$ to approximate the AOC