

# Neural networks

Spyros Samothrakis  
Research Fellow, IADS  
University of Essex

February 28, 2017

About

Linear function approximation with SGD

From linear regression to neural networks

Practical aspects

Conclusion

# ABOUT

- ▶ We will try to get a practical understanding of neural networks
  - ▶ The topic is massive
- ▶ The field has changed names a number of times
  - ▶ Connectionist systems
  - ▶ Neural networks
  - ▶ Deep learning
- ▶ They are all the same stuff, different name
  - ▶ Re-branding

# LINEAR FUNCTION APPROXIMATION

- ▶ One of the simplest ways to do a prediction
- ▶ In case we have a single variable/feature/co-variate, we are trying to learn
  - ▶  $\hat{f}_w(x) = w_1x + w_0$
- ▶  $w_0$  is often called the bias
- ▶  $x$  is a sample from the data
- ▶ Let's load some data

# DATA LOADING

```
df_linear = pd.DataFrame()

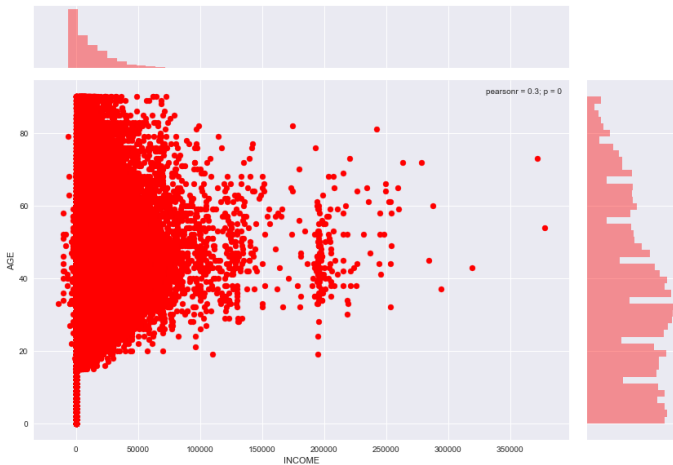
df_linear["AGE"] = df[["AGE"]].copy()
df_linear["INCOME"] = df[["INCOME" + str(i) for i in range(1,8)]].sum(axis = 1)

df_linear["YEARSCH"] = df[["YEARSCH"]].copy()
df_linear["ENGLISH"] = df[["ENGLISH"]].copy()
df_linear["FERTIL"] = df[["FERTIL"]].copy()
df_linear["YRSSERV"] = df[["YRSSERV"]].copy()
```

Is there another way we could have done the same?

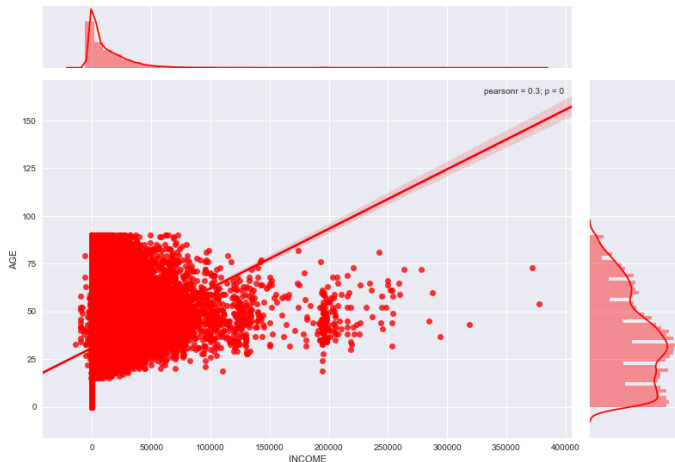
# VISUALISATION

```
g = sns.jointplot("INCOME", "AGE", data=df_linear, color = "r")
```



# VISUALISATION - LINEAR REGRESSION

```
g = sns.jointplot("INCOME", "AGE", data=df_linear, color = "r", kind="reg")
```



# STOCHASTIC GRADIENT DESCENT (SGD)

- ▶ Let's define a cost function  $J_w(x, y)$ , which signifies how far we are from the solution
- ▶ We have seen such cost functions before in terms of evaluating prediction

$$\blacktriangleright J_w(x, y) = \frac{1}{N} \sum_{i=1}^N \left( y^{(i)} - \hat{f}_w(x^{(i)}) \right)^2$$

- ▶ For just one example we have:

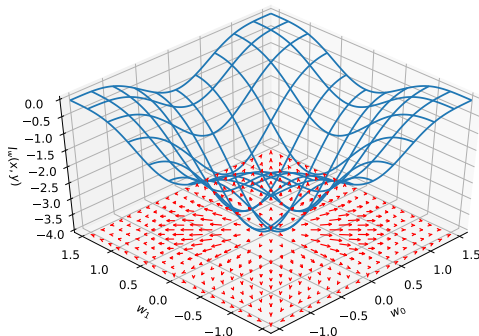
$$\blacktriangleright J_w(x^{(0)}, y^{(0)}) = \left( y^{(0)} - \hat{f}_w(x^{(0)}) \right)^2$$

$$w = w - \eta \cdot \nabla_w J_w(x, y)$$



# GRADIENTS

The gradient  $\nabla_w J_w(x, y)$  is the direction of steepest descent<sup>1</sup>



$$\nabla_w J_w(x, y) \equiv \left[ \frac{\partial J_w(x, y)}{\partial w_0}, \frac{\partial J_w(x, y)}{\partial w_1}, \dots, \frac{\partial J_w(x, y)}{\partial w_n} \right]$$

<sup>1</sup>[https://commons.wikimedia.org/wiki/File:Gradient\\_Visual.svg](https://commons.wikimedia.org/wiki/File:Gradient_Visual.svg)

# FIND THE GRADIENT

```
from sympy import Function, Symbol, latex, init_printing

y0,y1,y2 = symbols('y^(0),y^(1),y^(2)')
x0, x1, x2, x = symbols('x^(0), x^(1), x^(2), x')

f = (x*w1 + w0)

w0,w1 = symbols('w0, w1')

mse = ((f.subs(x, x0) - y0 )**2 )

mse.diff(w0)
mse.diff(w1)
```

$$\frac{\partial J_w(x,y)}{\partial w_0} = 2 \left( w_0 + w_1 x^{(0)} - y^{(0)} \right)$$
$$\frac{\partial J_w(x,y)}{\partial w_1} = 2x^{(0)} \left( w_0 + w_1 x^{(0)} - y^{(0)} \right)$$

## SOME SIMPLIFICATIONS

$$[w_0, w_1] = [w_0, w_1] - \eta \cdot \left[ \frac{\partial J_w(x, y)}{\partial w_0}, \frac{\partial J_w(x, y)}{\partial w_1} \right]$$

$$[w_0, w_1] = [w_0, w_1] - \eta \cdot \left[ 2 \left( w_0 + w_1 x^{(0)} - y^{(0)} \right), 2x^{(0)} \left( w_0 + w_1 x^{(0)} - y^{(0)} \right) \right]$$

$$[w_0, w_1] = [w_0, w_1] - \eta \cdot \left[ 2 \left( y^{(0)} - \hat{f}_w \left( x^{(i)} \right) \right), 2x^{(0)} \left( y^{(0)} - \hat{f}_w \left( x^{(i)} \right) \right) \right]$$

$$[w_0, w_1] = [w_0, w_1] - \eta \cdot \left[ \left( y^{(0)} - \hat{f}_w \left( x^{(i)} \right) \right), x^{(0)} \left( y^{(0)} - \hat{f}_w \left( x^{(i)} \right) \right) \right]$$

# MINI-BATCHES

- ▶ We used only one example
- ▶ This is called “online learning”
- ▶ But it’s common to use “mini-batches”
  - ▶ i.e., a number of samples together
- ▶ You would use every single sample
  - ▶ Often mini-batching much faster
  - ▶ Finding the right number of batches is tricky
  - ▶ Finding  $\eta$  is tricky

# WHAT IF WE HAD THREE WEIGHTS AND TWO FEATURES?

- Let's do the derivations

# SOME ANIMATIONS

<http://imgur.com/a/Hqolp>

<http://imgur.com/SmDARzn>

By Alec Reed

# CREATING HIGHER ORDER FEATURES

- ▶ What if the relationship between the features and the rewards is not linear?
- ▶ What if there is some other relationship?
- ▶ We can learn higher order features
- ▶ So for example, if we have  $x_0, x_1$ 
  - ▶ Superscripts are rows, subscripts are columns!
- ▶ We can create new features  $x_0^2, x_0x_1, x_1^2$

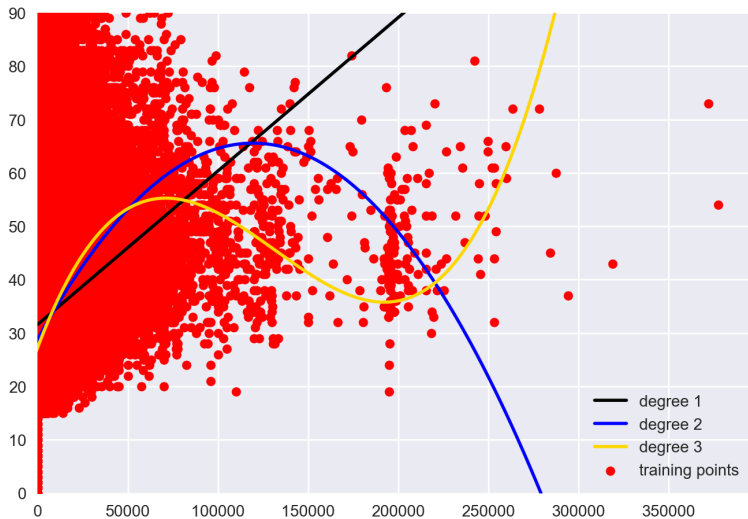
# REGRESSION WITH HIGHER ORDER FEATURES

```
from sklearn.preprocessing import PolynomialFeatures

degree = 1
model = make_pipeline(PolynomialFeatures(degree), StandardScaler(), SGDRegressor())
model.fit(X, y)
y_hat = model.predict(X)
```



## PLOT



# ADDING ANY OTHER KIND OF FEATURE

- ▶ How about we add features in the form
  - ▶  $\max(0, x_0)$
  - ▶  $\max(0.3, x_0)$
  - ▶  $\sin(x_0)$
  - ▶  $\sin(\max(0, x_0))$
- ▶ The list is endless
- ▶ Or, even better, *discover* those features

# BUILDING ABSTRACTIONS

- ▶ You rarely operate directly on the input space
  - ▶ People seem to be building abstract features
- ▶ We go from shapes and colours to forms to objects
- ▶ From primitive sounds to music
- ▶ Hierarchies of features extractors

# LEARNING FEATURES

- ▶ Let's revisit our example

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

- ▶ Let's add some depth using another function

$$\hat{f}_{w^{(0)}}(x) = w_1^{(0)} \hat{g}_w(x) + w_0^{(0)}$$

$$\hat{g}_{w^{(1)}}(x) = \max(w_1^{(1)} x + w_0^{(1)}, 0.3)$$

- ▶ The neuron above ( $g$ ) are known as a “leaky rectifier” neuron
  - ▶ But it's just a function - the neuron terminology comes from the original inspiration
  - ▶ Can you see why the name “deep learning” stuck?

# LEARNING

- Use the same error function to find errors

$$J_w(x^{(0)}, y^{(0)}) = \left(y^{(0)} - \hat{f}_w(x^{(i)})\right)^2$$

- Update each weight using the same method

$$w = w - \eta \cdot \nabla_w J_w(x, y)$$

$$\nabla_w J_w(x, y) \equiv \left[ \frac{\partial J_w(x, y)}{\partial w_0}, \frac{\partial J_w(x, y)}{\partial w_1}, \dots, \frac{\partial J_w(x, y)}{\partial w_n} \right]$$

- Use the chain rule to calculate partial derivatives

$$\frac{d}{dx} [f(g(x))] = f'(g(x))g'(x)$$

# CONVEXITY

- ▶ We are going downhill
- ▶ What if the error surface is smooth, it's all good
- ▶ But what if the error surface is rugged mess with ups and downs?
- ▶ Local optima are not global optima
- ▶ Thus, NN training often somewhat unstable

# THEANO, TENSORFLOW

- ▶ As you understand doing all these calculations by hand is tedious
- ▶ Every new type of neuron or layer you invent would require a new gradient calculation
- ▶ Auto-differentiation
  - ▶ Theano
  - ▶ Tensorflow
- ▶ You just describe your network/graph
- ▶ The rest is done automatically by the machine

# BUILDING A NETWORK

- ▶ There are multiple packages for neural networks
- ▶ I think the most popular is Keras
- ▶ Though a massive number of people are using Lasagne



# KERAS API

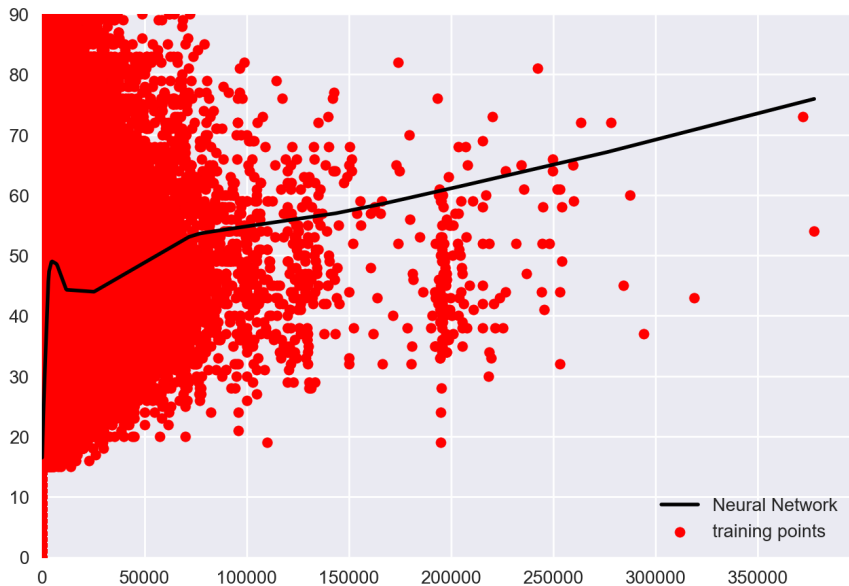
```
inputs = Input(shape=(1,))

x = Dense(64)(inputs)
x = PReLU()(x) # Non-linearity
x = Dense(64)(x)
x = PReLU()(x) # Non-linearity
predictions = Dense(1, activation='linear')(x)

model = Model(input=inputs, output=predictions)
model.compile(optimizer='adam', loss='mse')

model.fit(...)
```

# OUTPUT PLOT



# INPUT LAYER

```
inputs = Input(shape=(22,))
```

- ▶ The first layer
- ▶ Does not contain any neurons
  - ▶ Terminology changes between authors
- ▶ “Number of columns”
- ▶ “Number of features”

# HIDDEN LAYERS

```
x = PReLU()(x) # Non-linearity  
x = Dense(64)(x)  
x = PReLU()(x) # Non-linearity
```

- ▶ “Dense” layers specify the number of neurons
- ▶ You need to follow them up with a non-linear transformation
- ▶ Alternately you can pass this as parameter “activation”
- ▶ Non-linearity is often called the “activation function”

# OUTPUT LAYER

```
predictions = Dense(1, activation='linear')(x)
model = Model(input=inputs, output=predictions)
```

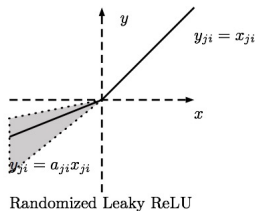
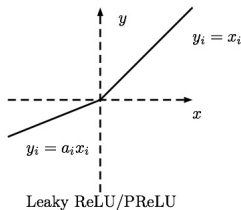
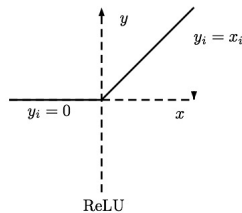
- ▶ The output layer
- ▶ A normal Dense layer
- ▶ “Number of outputs”
- ▶ You can predict multiple outputs at once

# COST (OR OBJECTIVE) FUNCTIONS

- ▶ You can do both regression and classification with neural networks
- ▶ All it changes is the output layer and the objective function
- ▶ Categorical cross-entropy for classification
  - ▶ You need to convert your outputs to one-hot encoding
- ▶ Mean squared error for prediction

# NEURON TYPES

- Also called activation functions or non-linearities



2

<sup>2</sup><http://lamda.nju.edu.cn/weixs/project/CNNTricks/imgs/relufamily.png>

# LEARNING ALGORITHMS

- ▶ We have just seen SGD, which forms the basis of every modern learning algorithm for NNs
  - ▶ It will be dethroned eventually, but nobody knows from what
- ▶ SGD does not take into account specific information about individual weights
  - ▶ Not the curvature of the search space
- ▶ There are methods that auto-adapt the learning rate per individual weight
  - ▶ Adam
  - ▶ RMSProp
- ▶ Adam should be your default option



# REGULARISATION

- ▶ Overfitting is a massive problem - with enough variables you can fit anything
- ▶  $\ell_1$ : Turn your weights down to zero
- ▶  $\ell_2$ : Make all your weights small
- ▶ You can define the regularisation strength
- ▶ Dropout:
  - ▶ Remove some of the weights during training uniform random
  - ▶ During testing/using put use everything, but toned down
  - ▶ Implemented as a distinct layer

## (BATCH) NORMALISATION

- ▶ Implemented as a different layer
- ▶ Covariate shift impact learning negatively
- ▶ Normalise the weights of each layer

$$\hat{w} = \frac{w - \mu}{\sigma}$$

$$y = \gamma \hat{w} + \beta$$

- ▶  $\gamma$  and  $\beta$  are special types of parameters, to be learned as well
  - ▶ They scale and shift a bit
- ▶ Stops “internal covariate shift”
  - ▶ Accelerates learning

## WEIGHT SHARING / MULTIPLE OUTPUTS

- ▶ You can create pathways for different things that share the same weights
- ▶ Thus imposing stronger regularisation
  - ▶ Use same features for different tasks
- ▶ Very effective method of generalising
- ▶ You can use different pathways within the same network

# METRICS/ CALLBACKS

- ▶ When should you stop training?
  - ▶ After  $i$  iterations over the whole dataset
  - ▶ After the validation error stops improving
- ▶ Save best validation model
  - ▶ ... and use this for testing
- ▶ Reduce learning rates, log etc
- ▶ “Callbacks”

# CREATING CUSTOM LAYERS

3

```

from keras import backend as K
from keras.engine.topology import Layer
import numpy as np

class MyLayer(Layer):
    def __init__(self, output_dim, **kwargs):
        self.output_dim = output_dim
        super(MyLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Create a trainable weight variable for this layer.
        self.W = self.add_weight(shape=(input_shape[1], self.output_dim),
                                initializer='random_uniform',
                                trainable=True)
        super(MyLayer, self).build() # Be sure to call this somewhere!

    def call(self, x, mask=None):
        return K.dot(x, self.W)

    def get_output_shape_for(self, input_shape):
        return (input_shape[0], self.output_dim)

```

---

<sup>3</sup><https://keras.io/layers/writing-your-own-keras-layers/>

# CONCLUSION

- ▶ We have touched upon neural networks
- ▶ It's a really hot topic right now
- ▶ Requires a bit of dedication
- ▶ New algorithms are coming out every day