# Basis Expansion & Splines

**Important Topics:**

Smoothing Functions

**Basis Functions:**

Basically a layer of abstraction in between the feature and the model that transforms the features in a certain way.

Polynomial and piecewise constant regression models are special cases of basis functions.

For polynomial regression, the basis functions are
$b_j(x_i) = x_i^j$,

For piecewise constant functions they are
$b_j(x_i) = I(x_j \leq x_i \leq c_{j+1})$

Wavelets or Fourier series could also be used to construct basis functions.

Examples:

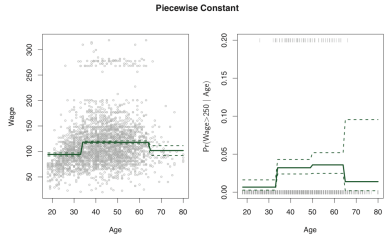
Basis Function with logistic regression:

$$\Pr(y_i > 250|x_i) = \frac{\exp(\beta_0 + \beta_1 C_1(x_i) + \ldots + \beta_K C_K(x_i))}{1 + \exp(\beta_0 + \beta_1 C_1(x_i) + \ldots + \beta_K C_K(x_i))}$$

Basis function with linear least squares:

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \beta_3 b_3(x_i) + \ldots + \beta_K b_K(x_i) + \epsilon_i.$$

Graphical Representation of both:




**Piecewise Polynomials:**

Instead of fitting a high-degree polynomial over the entire range of X, piecewise polynomial regression involves fitting separate low-degree polynomials over different regions of X.

$$y_i = \begin{cases} \beta_{01} + \beta_{11} x_i + \beta_{21} x_i^2 + \beta_{31} x_i^3 + \epsilon_i & \text{if } x_i < c; \\ \beta_{02} + \beta_{12} x_i + \beta_{22} x_i^2 + \beta_{32} x_i^3 + \epsilon_i & \text{if } x_i \geq c. \end{cases}$$

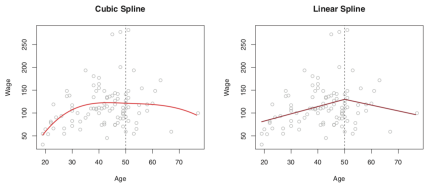
Disadvantages: The function is discontinuous.

**Constraints and Splines:**

Each constraint that we impose on the piecewise cubic polynomials effectively frees up one degree of freedom, by reducing the complexity of the resulting piecewise polynomial fit.

Example: Eight degrees of freedom (since it has 4 parameters), but in imposed three constraints (continuity, continuity of the first derivative, and continuity of the second derivative) and so are left with five degrees of freedom.

Cubic spline with K knots used a total of 4 + k degrees of freedom.




The general definition of a degree-d spline is that it is a piecewise degree-d polynomial, with continuity in derivatives up to degree d – 1 at each knot.

**Spline Basis Representation:**

A cubic spline with K knots can be fit with least squares and represented as

$$y_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \cdots + \beta_{K+3} b_{K+3}(x_i) + \epsilon_i,$$

The most direct way to represent a cubic spline using (7.9) is to start off with a basis for a cubic polynomial—namely, x,x2,x3—and then add one truncated power basis function per knot.

$$h(x, \xi) = (x - \xi)_+^3 = \begin{cases} (x - \xi)^3 & \text{if } x > \xi \\ 0 & \text{otherwise,} \end{cases}$$

Example:

In other words, in order to fit a cubic spline to a data set with K knots, we perform least squares regression with an intercept and 3 + K predictors, of the form
$X, X^2, X^3, h(X, \xi_1), h(X, \xi_2) \ldots h(X, \xi_k)$

where ξ1,...,ξK are the knots. This amounts to estimating a total of K + 4 regression coefficients; for this reason, fitting a cubic spline with K knots uses K+4 degrees of freedom.

A natural spline is a regression spline with additional boundary constraints: the function is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot)

one option is to place more knots in places where we feel the function might vary most rapidly, and to place fewer knots where it seems more stable.

Compared to Polynomial regression, splines yield superior results. To produce flexible fits, splines introduce flexibility by increasing the number of knots but keeping the degree fixed.

**Smoothing Splines:**

$$\sum_{i=1}^{n}(y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt$$

where $\lambda$ is a non-negative smoothing function. the term λ g (t) dt is a penalty term that penalizes the variability in g.

If g is very smooth, then g′(t) will be close to constant and $\int g''(t)^2 dt$ will take on a small value.

λ = ∞ -> underfit, λ = 0 -> overfit .

Smoothing natural cubic spline function is a piecewise cubic polynomial with knots at the unique values of x1,...,xn, and continuous first and second derivatives at each knot. Furthermore, it is linear in the region outside of the extreme knots.

It is possible to show that as λ increases from 0 to ∞, the effective degrees of freedom, which we write dfλ, decrease from n to 2

**Choosing a lambda:**

Although a smoothing spline has n parameters and hence n nominal degrees of freedom, these n parameters are heavily constrained or shrunk down.

dfλ is a measure of the flexibility of the smoothing spline—the higher it is, the more flexible (and the lower-bias but higher-variance) the smoothing spline.

$$df_\lambda = \sum_{i=1}^{n}\{\mathbf{S}_\lambda\}_{ii},$$

$S_\lambda$ is the nXn matrix vector of fitted values when applying a smoothing spline to the data. $g$ is a n-vector containing the fitted values of the smoothing spline at the training points $x_1 \ldots x_n$ such that $\hat{\mathbf{g}}_\lambda = \mathbf{S}_\lambda \mathbf{y},$

Leave-one-out cross-validation error (LOOCV) can be computed very efficiently for smoothing splines, with essentially the same cost as computing a single fit, using the following formula: $\mathrm{RSS}_{cv}(\lambda) = \sum_{i=1}^{n}(y_i - \hat{g}_\lambda^{(-i)}(x_i))^2 = \sum_{i=1}^{n}\left[\frac{y_i - \hat{g}_\lambda(x_i)}{1 - \{\mathbf{S}_\lambda\}_{ii}}\right]^2$

**Local Regression:**

Span: The span plays a role like that of the tuning parameter λ in smoothing splines: it controls the flexibility of the non-linear fit. The smaller the value of s, the more local and wiggly will be our fit; alternatively, a very large value of s will lead to a global fit to the data using all of the training observations.

# Kernel Density Estimation & Convex Estimation

Probability that a point x lies in region R:

$$p(\mathbf{x}) = \frac{K}{NV}.$$

where N is the number of observations, K is the number of points in region R and V is the volume of region R.

Kernel function is defined by:

$$k(\mathbf{u}) = \begin{cases} 1, & |u_i| \leqslant 1/2, \\ 0, & \text{otherwise} \end{cases} \qquad i = 1, \ldots, D,$$

$k\left((x - x_n)/h\right)$ will be one if the data point $x_n$ lies inside a cube of side $h$ centred on $h$, and zero otherwise.

The total number of data points lying inside this cube will therefore be:

$$K = \sum_{n=1}^{N} k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right).$$

Combining the above equation with the first one, we get:

$$p(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{h^D}k\left(\frac{\mathbf{x} - \mathbf{x}_n}{h}\right)$$

To combat discontinuity, we apply gaussian smoothing to p(x) to get: $p(\mathbf{x}) = \frac{1}{N}\sum_{n=1}^{N}\frac{1}{(2\pi h^2)^{1/2}}\exp\left\{-\frac{\|\mathbf{x} - \mathbf{x}_n\|^2}{2h^2}\right\}$

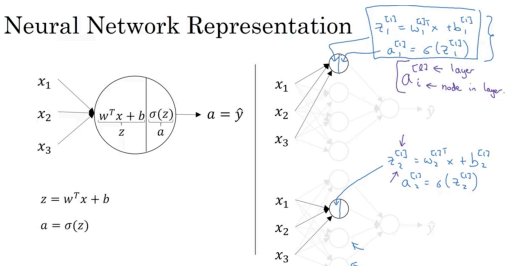
**Local Regression Algorithm:**

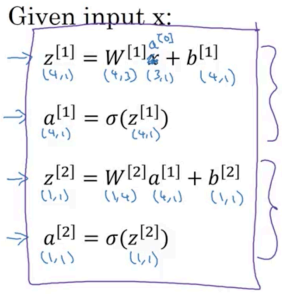
**Algorithm 7.1** *Local Regression At $X = x_0$*

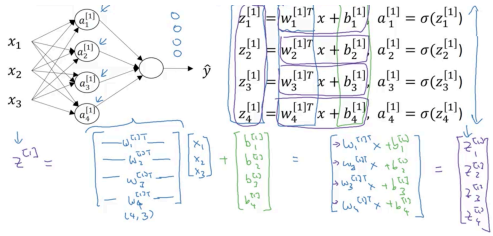
1. Gather the fraction $s = k/n$ of training points whose $x_i$ are closest to $x_0$.
2. Assign a weight $K_{i0} = K(x_i, x_0)$ to each point in this neighborhood, so that the point furthest from $x_0$ has weight zero, and the closest has the highest weight. All but these $k$ nearest neighbors get weight zero.
3. Fit a *weighted least squares regression* of the $y_i$ on the $x_i$ using the aforementioned weights, by finding $\hat{\beta}_0$ and $\hat{\beta}_1$ that minimize
$$\sum_{i=1}^{n} K_{i0}(y_i - \beta_0 - \beta_1 x_i)^2. \qquad (7.14)$$
4. The fitted value at $x_0$ is given by $\hat{f}(x_0) = \hat{\beta}_0 + \hat{\beta}_1 x_0$.

# Neural Networks

**Basics:**



For all training examples:

```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

**Activation Functions:**

Sigmoid: $\frac{1}{1+e^{-z}}$ | Generally inferior to tanh, useful for binary output. Range: [0 - 1]

tanh: $\frac{e^z - e^{-z}}{e^z + e^{-z}}$ | Superior to sigmoid because the mean ends up being centered around 0. | Gradient Descent is slow when z is super high or super low as the gradients are close to 0. Range: [-1 - 1]

ReLu: $a = max(0, z)$ | Is efficient because of the constant gradient. Problem is, derivative at 0 is undefined. Leaky ReLu fixes that problem. Range: [0 - max(z)]

**Backprop:**

$dz^{[2]} = a^{[2]} - y$

$dW^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

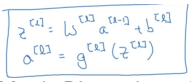
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(z^{[1]})$

$dW^{[1]} = dz^{[1]} x^T$

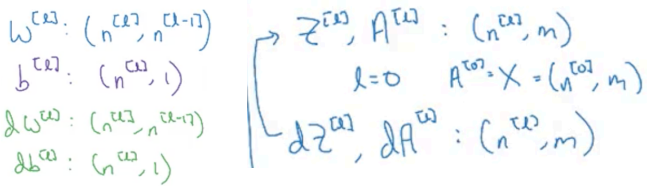
$db^{[1]} = dz^{[1]}$

**Forward Prop for L Layers:**

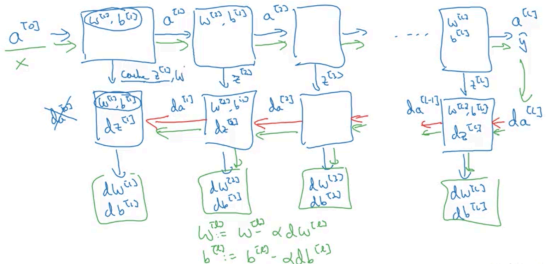
General formula for computing w and b



Matrix Dimensions: (Non-Vectorized vs Vectorized)



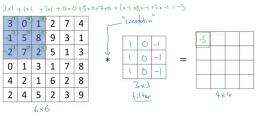
Learning Visualized:



**CNN:**

nxn * fxf = oxo

oxo = n-f+1 * n-f+1



Padding: +ve's: edge info is not lost | 'Valid' is no padding | 'Same' = n+2p - f+1 = n; p = (f-1)/2

Stride: (oxo) :n + 2p - f)/s + 1 * (n + 2p - f)/s + 1 [s is stride val]

Maxpool: