STAT406- Methods of Statistical Learning Lecture 19

Matias Salibian-Barrera

UBC - Sep / Dec 2017

TO COMPLETE YOUR REGISTRATION, PLEASE TELL US WHETHER OR NOT THIS IMAGE CONTAINS A STOP SIGN:

NO YES

ANSWER QUICKLY—OUR SELF-DRIVING CAR IS ALMOST AT THE INTERSECTION.

SO MUCH OF "AI" IS JUST FIGURING OUT WAYS TO OFFLOAD WORK ONTO RANDOM STRANGERS.

https://xkcd.com/1897/

Boosting

- AdaBoost uses the following loss function

$$L\left(y, G(\mathbf{x})\right) \;=\; \exp\left(-y\, G\left(\mathbf{x}\right)\right)$$

- At the j-th iteration we have

$$\arg\min_{\beta,\gamma} \sum_{i=1}^{n} \exp\Big(-y_i \left(f_{j-1}(\mathbf{x}_i) + \beta f(\mathbf{x}_i, \gamma)\right)\Big)$$

$$\arg\min_{\beta,\gamma} \sum_{i=1}^{n} w_i^{(l-1)} \exp\Big(-\beta\, y_i f(\mathbf{x}_i, \gamma)\Big)$$

Boosting

- For any $\beta > 0$ the solution is the classifier $f(\mathbf{x}, \gamma)$ that minimizes

$$\sum_{y_i \neq f(\mathbf{x}_i, \gamma)} w_i^{(j-1)} = \sum_{i=1}^{n} w_i^{(j-1)} I\left(y_i \neq f(\mathbf{x}_i, \gamma)\right)$$

  which is a weighted missclassification error

Boosting

- Similarly we obtain

$$\beta_j = \frac{1}{2}\log\left(\frac{1-e_j}{e_j}\right)$$

where

$$e_j = \sum_{i=1}^{n} w_i^{(j-1)} I\left(y_i \neq f(\mathbf{x}_i, \gamma_j)\right) \Big/ \sum_{i=1}^{n} w_i^{(j-1)}$$

Boosting

- We then update

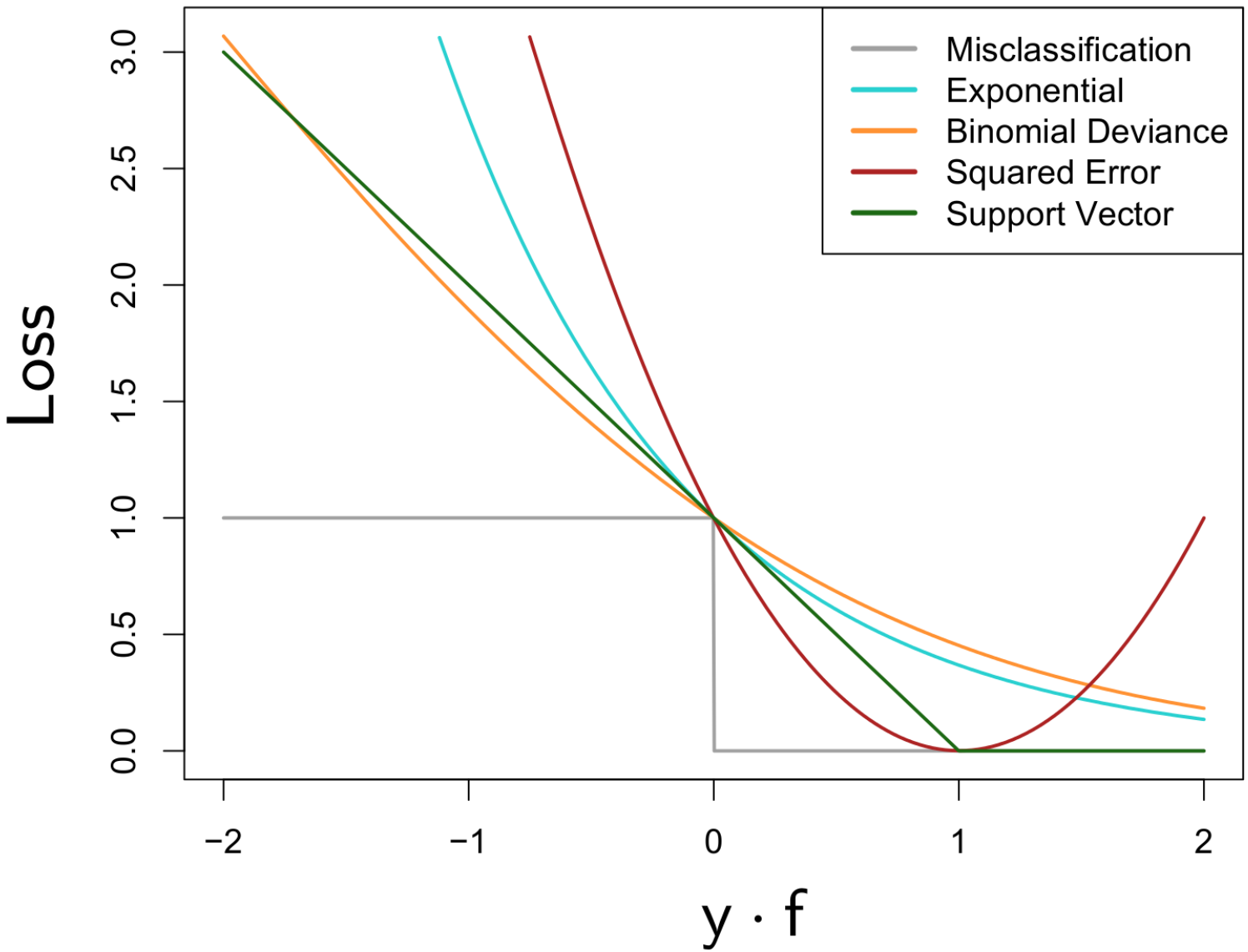
$$f_j(\mathbf{x}) \;=\; f_{j-1}(\mathbf{x}) + \beta_j\, f(\mathbf{x}, \gamma_j)$$

and hence

$$w_i^{(j+1)} \;=\; w_i^{(j)}\, \exp\left(-\beta_j\, y_i\, f(\mathbf{x}_i, \gamma_j)\right)$$
$$= \exp\left(-\beta_j\right)\, w_i^{(j)}\, \exp\left(-\alpha_j\, I\left(y_i \neq f(\mathbf{x}_i, \gamma_j)\right)\right)$$

where $\alpha_j = 2\,\beta_j$

Loss functions



Boosting

- The exponential loss penalizes misclassifications more than it approves correct classifications
- In particular, severe mistakes are very costly
- but the benefit of correct calls changes much more slowly

Boosting

- One can show that the "population" solution

$$\arg\min_{G(\mathbf{x})} E_{Y|\mathbf{X}=\mathbf{x}}\left[\exp\left(-Y\,G(\mathbf{x})\right)\right] \;=$$

$$= \frac{1}{2}\,\log\left(\frac{P(Y=1|\,\mathbf{X}=\mathbf{x})}{P(Y=-1|\,\mathbf{X}=\mathbf{x})}\right)$$

- The deviance loss also has the same "target" solution but grows slower - (so what?)

Boosting

- The shape of the exponential loss means that even if we have perfect classification for the training data, the objective function

$$\frac{1}{n}\sum_{i=1}^{n} L\left(y_i, G(\mathbf{x}_i)\right)$$

  may not have reached its minimum

- Thus the iterations continue...

Boosting

- Since we know what this method is estimating

$$\frac{1}{2}\log\left(\frac{P(Y=1|\mathbf{X}=\mathbf{x})}{P(Y=-1|\mathbf{X}=\mathbf{x})}\right)$$

... and we know what type of functions is attempting to use

$$\sum_{j=1}^{K}\beta_j f(\mathbf{x},\gamma_j) = \frac{1}{2}\log\left(\frac{P(Y=1|\mathbf{X}=\mathbf{x})}{P(Y=-1|\mathbf{X}=\mathbf{x})}\right)$$

Boosting

- ... we can understand when it works and when it may not work
- Note that the class of base classifiers $f(\mathbf{x}, \gamma)$ determines the type of log odds ratio we can model
- In particular, when using trees, the number of leaves (terminal nodes) determines the degree of interaction among the features that it may be able to capture

Gradient Boosting

- We are trying to find

$$\hat{\mathbf{f}} = \arg\min_{\mathbf{f}\in\mathbb{R}^n} \sum_{i=1}^{n} L(y_i, f_i)$$

- Many numerical optimization methods compute $\hat{\mathbf{f}}$ iteratively

$$\hat{\mathbf{f}} = \sum_{\ell=1}^{K} \mathbf{a}_\ell$$

where $\mathbf{a}_\ell \in \mathbb{R}^n$.

Gradient Boosting

- For example, gradient descent methods

$$\mathbf{a}_\ell = -\lambda_\ell \nabla L(\mathbf{f})|_{\mathbf{f}=\mathbf{f}_{\ell-1}}$$

where

$$\mathbf{f}_j = \begin{pmatrix} f_j(\mathbf{x}_1) \\ f_j(\mathbf{x}_2) \\ \vdots \\ f_j(\mathbf{x}_n) \end{pmatrix} \in \mathbb{R}^n$$

Gradient Boosting

- The difference is that in

$$\boldsymbol{\Theta}_j = \arg\min_{\boldsymbol{\Theta}} \sum_{i=1}^{n} L\left(y_i, f_{j-1}(\mathbf{x}_i) + T(\mathbf{x}_i, \boldsymbol{\Theta})\right)$$

the components of the "update" vector

$$\begin{pmatrix} T(\mathbf{x}_1, \boldsymbol{\Theta}) \\ T(\mathbf{x}_2, \boldsymbol{\Theta}) \\ \vdots \\ T(\mathbf{x}_n, \boldsymbol{\Theta}) \end{pmatrix}$$

are constrained to result from a single tree

Gradient Boosting

- In addition, our goal is to "generalize" $\sum_{j=1}^{K} T(\mathbf{x}_j, \mathbf{\Theta})$ to other $\mathbf{x}$'s
- Gradients, on the other hand, are much easier to compute

Gradient Boosting

- This suggests the following, at each step

$$\arg\min_{\Theta} \sum_{i=1}^{n} \left(-g_{i,\ell} - T\left(\mathbf{x}_i, \Theta\right)\right)^2$$

where $g_{i,\ell}$ is the i-th element of the gradient evaluated at the ℓ-th step:

$$\nabla L\left(\mathbf{f}\right)\big|_{\mathbf{f}=\mathbf{f}_{\ell-1}}$$

Gradient Tree Boosting

- Initialize

$$\mathbf{f}_0 = \arg\min_{\gamma \in \mathbb{R}} \sum_{i=1}^{n} L(y_i, \gamma)$$

- For $j = 1, 2, ..., K$
- Let

$$g_{i,j} = -\left.\frac{\partial L(y_i, f)}{\partial f}\right|_{f=f_{j-1}(\mathbf{x}_i)} \qquad i = 1, 2, \ldots, n$$

- Fit a regression tree to the "responses" $g_{1,j}, g_{2,j}, \ldots, g_{n,j}$, obtaining the regions $R_{1,j}, \ldots, R_{M_j,j}$

Gradient Tree Boosting

- For $h = 1, 2, ..., M_j$ find the constants

$$\gamma_{h,j} = \arg\min_{\gamma} \sum_{\mathbf{x}_i \in R_{h,j}} L(y_i, f_{j-1}(\mathbf{x}_i) + \gamma)$$

- Let

$$f_j(\mathbf{x}) = f_{j-1}(\mathbf{x}) + \sum_{h=1}^{M_j} \gamma_{h,j} \mathbf{I}\left(\mathbf{x} \in R_{h,j}\right)$$

Gradient Tree Boosting

- Shrinkage:

$$f_j(\mathbf{x}) = f_{j-1}(\mathbf{x}) + \tau \sum_{h=1}^{M_j} \gamma_{h,j} \mathbf{I}\left(\mathbf{x} \in R_{h,j}\right)$$

  where $\tau \in (0, 1)$

- τ is the "rate of learning"
- Small values of τ require more iterations (K)
- This approach works very well in practice. More work is needed.

Neural Networks

- Neural networks are flexible regression models
- Hybrid (or particular case) of non-parametric regression and projection-pursuit
- Versatile, computationally very costly and fragile

Neural Networks

- Fitting them can almost be considered an art...
- Recently they've made a come back under the umbrella of "deep learning"

Neural Networks

- Neural networks (NN) build flexible regression models
- They use an ordered sequence of unobserved units that are transformed linear combinations of units appearing in previous levels of the network
- We'll focus on single-layer NNs (to simplify the discussion)

Neural Networks

- Let $\mathbf{X} \in \mathbb{R}^p$ be a generic vector of explanatory variables
- Build a new set of explanatory variables

$$Z_m = \sigma\left(\alpha_{0,m} + \boldsymbol{\alpha}'_m \mathbf{X}\right), \quad m = 1, 2, \ldots, M$$

(These form the hidden layer)

Neural Networks

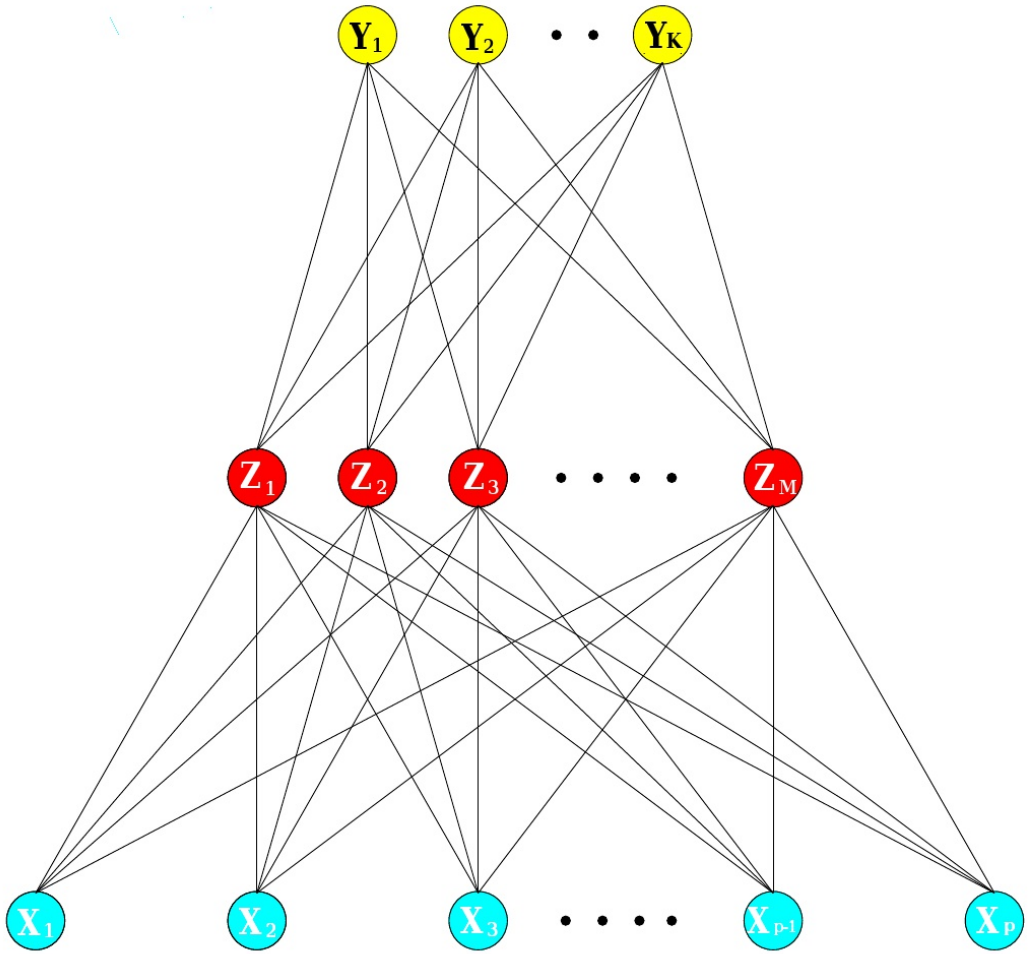
- The output variables are then linear combinations of the Z_m's

$$T_k = \beta_{0,k} + \boldsymbol{\beta}_k' \mathbf{Z}, \quad k = 1, 2, \ldots, K$$

- The output variables may themselves be transformed again

$$f_k = f_k(\mathbf{X}) = g_k(\mathbf{T})$$

Neural Networks



Neural Networks

- The "activation function" is usually

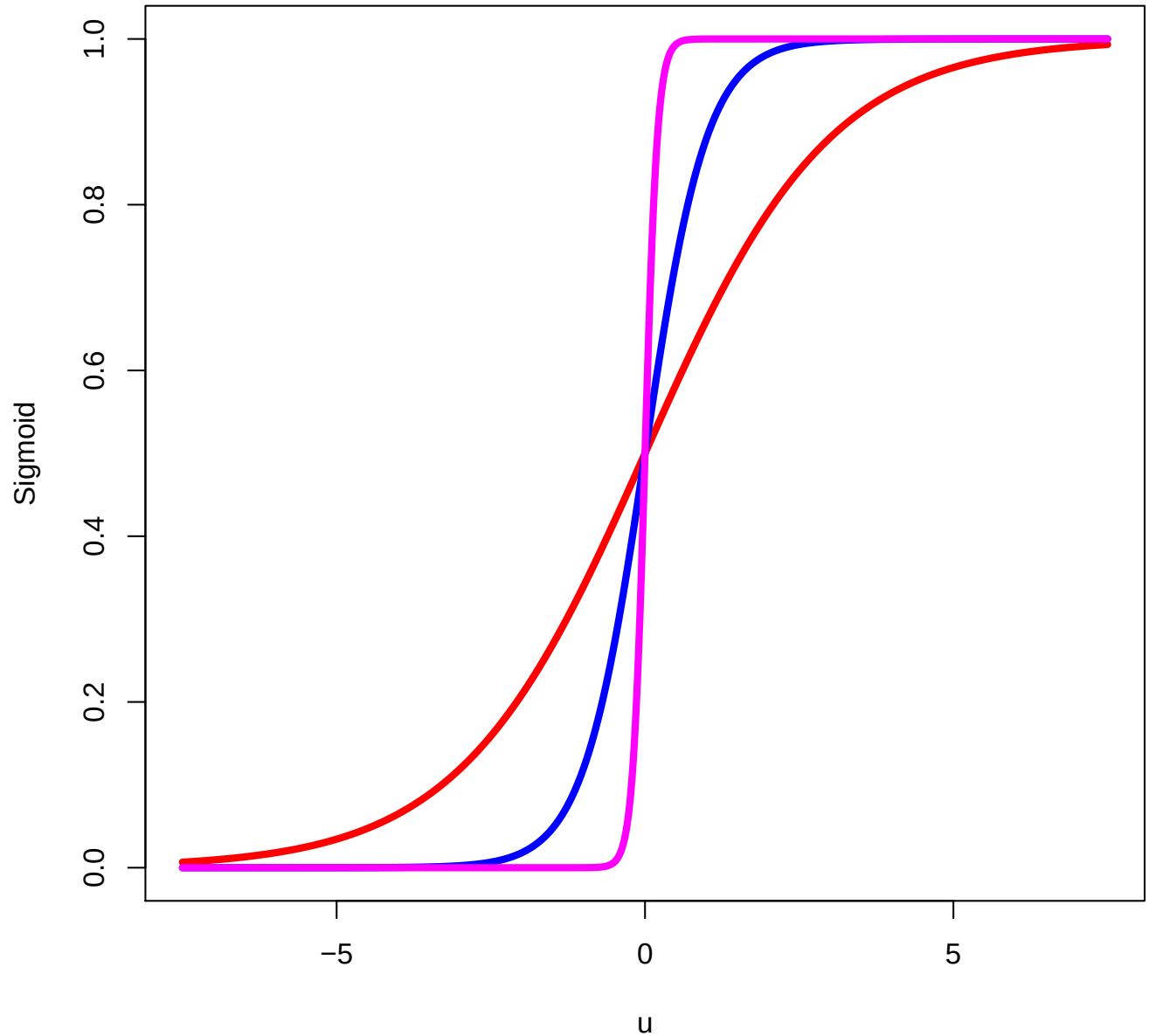
$$\sigma(u) = \frac{1}{1+\exp(-u)} = \frac{\exp(u)}{1+\exp(u)}$$

- For continuous responses we set $K = 1$ and $g_k(\mathbf{T}) = T_k$
- For categorical responses K is the number of classes and

$$g_k(\mathbf{T}) = \frac{\exp(T_k)}{\sum_{s=1}^{K}\exp(T_s)}$$

(aka "soft-max" outputs)

“Activation function”



Neural Networks

- How do we estimate ("learn") the α's and β's?
- For continuous responses

$$\min_{\alpha,\beta} \sum_{i=1}^{n} (y_i - f_1(\mathbf{x}_i))^2$$

$$\min_{\alpha,\beta} \sum_{i=1}^{n} \left(y_i - \beta_0 - \beta' \mathbf{Z}\right)^2$$

$$\min_{\alpha,\beta} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{s=1}^{M} \beta_s \sigma\left(\alpha_{0,s} + \alpha'_m \mathbf{x}_i\right) \right)^2$$

Neural Networks

- For categorical responses we use the deviance (cross-entropy) function

$$\min_{\alpha,\beta} \sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k} \log(f_k(\mathbf{x}_i))$$

Neural Networks

- Soft-max outputs and cross-entropy loss = logistic regression model on the variables in the hidden layer + MLE estimation
- By adding variables in the hidden layer the model becomes more flexible

Neural Networks

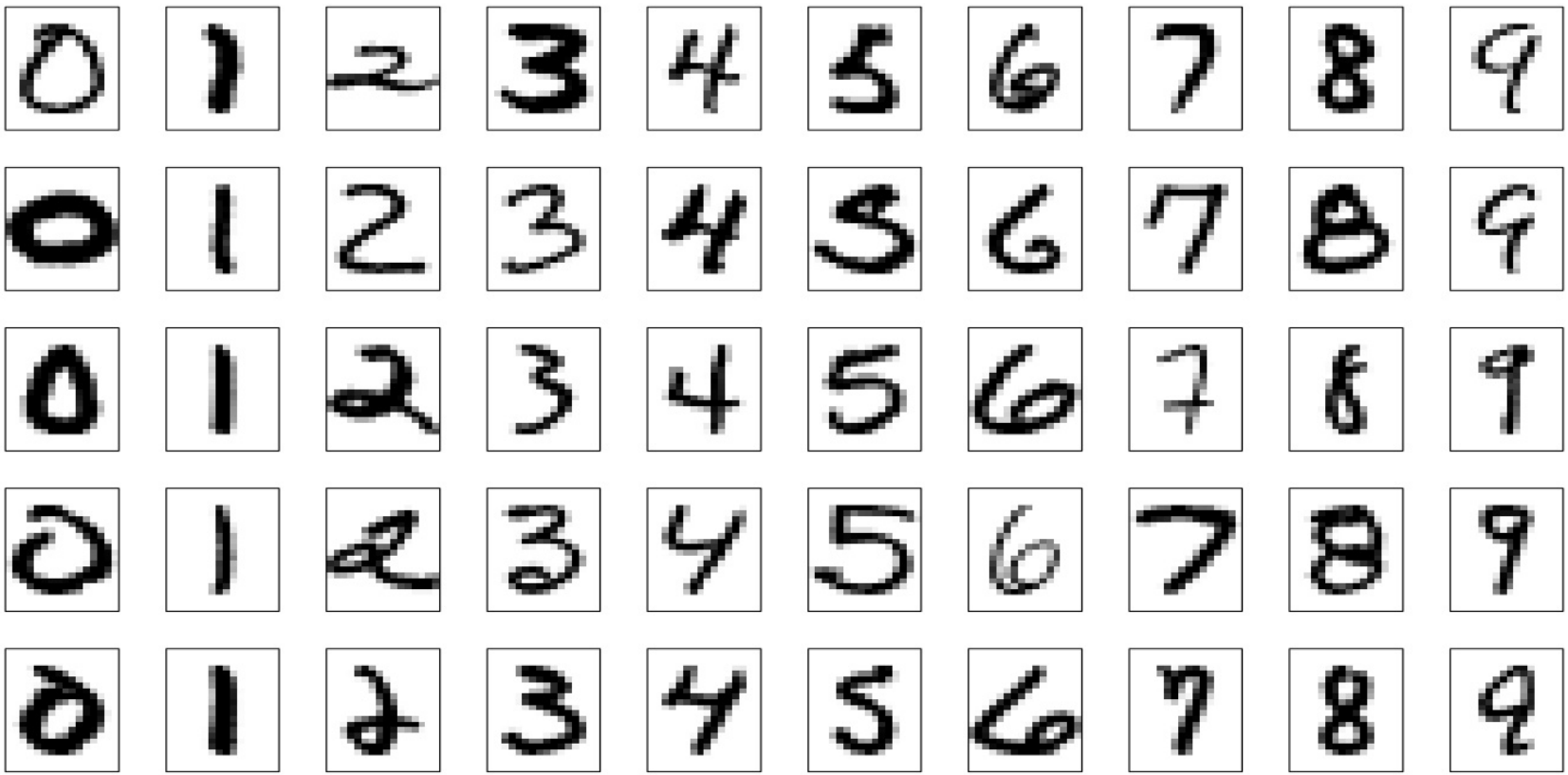
- Overfitting / regularization

$$\min_{\alpha,\beta} \sum_{i=1}^{n}\sum_{k=1}^{K} y_{i,k} \log(f_k(\mathbf{x}_i)) + \lambda P(\alpha,\beta)$$

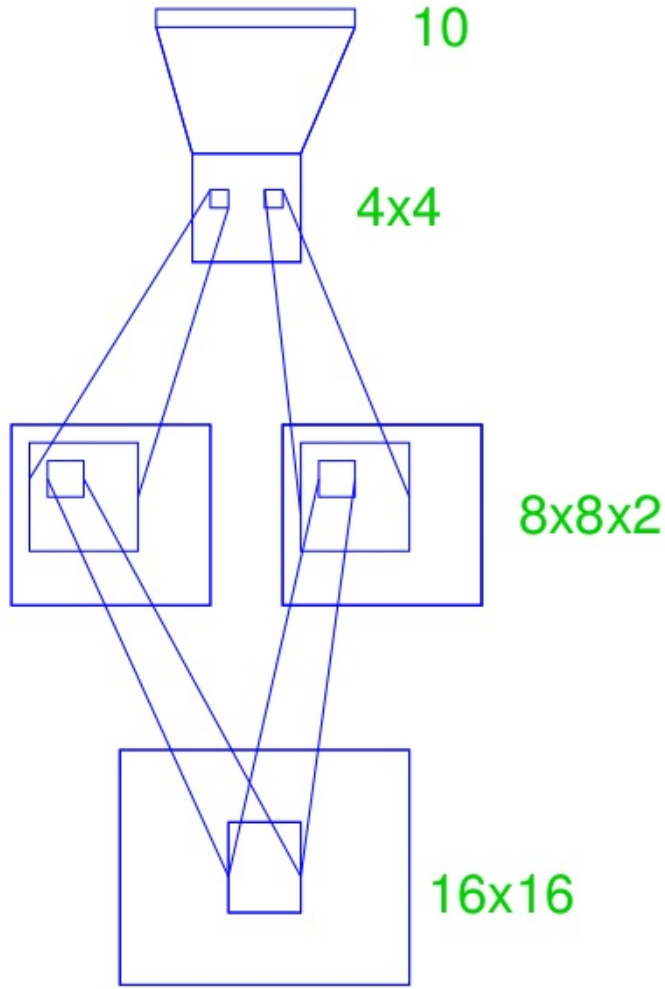
$$P(\alpha,\beta) = \|\alpha\|_2^2 + \|\beta\|_2^2$$

- A.K.A. as “weight decay”
- Since random starts are needed, the scale of the input variables becomes a potentially important issue

Feature “discovery”



Feature “discovery”



Net-4

Shared Weights

Neural Networks - ISOLET

```
> # ISOLET EXAMPLE
>
> # 3 and 26 "C" and "Z"
>
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=1, decay=0, maxit=1500, MaxNWts=2000)
# weights:  620
initial  value 350.425020
iter  10 value 41.176789
...
iter 120 value 6.482733
final  value 6.482722
converged
> a1$value
[1] 6.482722
>
>
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0.002083333
>
```

Neural Networks - ISOLET

```
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=1, decay=0, maxit=1500, MaxNWts=2000)
# weights:
initial  value 336.934868
iter  10 value 157.630462
...
iter 150 value 16.164762
final  value 16.164753
converged
> a2$value
[1] 16.16475
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0.00625
>
> b1 <- predict(a1, newdata=x.te, type='class') #, type='raw')
> mean(b1 != x.te$V618)
[1] 0.03333333
>
> b2 <- predict(a2, newdata=x.te, type='class') #, type='raw')
> mean(b2 != x.te$V618)
[1] 0.025
```

Neural Networks - ISOLET

```
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0, maxit=1500, M
# weights:  1858
initial  value 334.262969
...
iter  90 value 6.482739
final  value 6.482738
converged
>
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0, maxit=1500, M
# weights:  1858
initial  value 348.931860
...
iter  30 value 0.001012
final  value 0.000091
converged
```

Neural Networks - ISOLET

```
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0.002083333
>
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0
>
> b1 <- predict(a1, newdata=x.te, type='class') #, type='r
> mean(b1 != x.te$V618)
[1] 0.03333333
>
> b2 <- predict(a2, newdata=x.te, type='class') #, type='r
> mean(b2 != x.te$V618)
[1] 0.04166667
```

Neural Networks - ISOLET

```
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=6, decay=0.05, maxi
# weights:  3715
...
iter 110 value 4.777807
final  value 4.777806
converged
>
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=6, decay=0.05, maxi
# weights:  3715
...
iter 260 value 4.172023
final  value 4.172023
converged
```

Neural Networks - ISOLET

```
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0
>
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0
>
> b1 <- predict(a1, newdata=x.te, type='class') #, t
> mean(b1 != x.te$V618)
[1] 0.008333333
>
> b2 <- predict(a2, newdata=x.te, type='class') #, t
> mean(b2 != x.te$V618)
[1] 0.008333333
```

Neural Networks - ISOLET

```
> ### More letters
> lets <- c(3, 7, 9, 26)
> LETTERS[lets]
[1] "C" "G" "I" "Z"
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=1, decay=0, maxit=1500, MaxN
# weights:  626
...
iter 860 value 6.482741
final  value 6.482739
converged
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=1, decay=0, maxit=1500, MaxN
# weights:  626
...
iter  40 value 789.912166
final  value 789.900872
converged
```

Neural Networks - ISOLET

```
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0.001041667
>
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0.5010417
>
> b1 <- predict(a1, newdata=x.te, type='class') #, type='r
> mean(b1 != x.te$V618)
[1] 0.4666667
>
> b2 <- predict(a2, newdata=x.te, type='class') #, type='r
> mean(b2 != x.te$V618)
[1] 0.525
```

Neural Networks - ISOLET

```
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0, r
# weights:  1870
...
iter 410 value 27.422499
final  value 27.422441
converged
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0, r
...
iter 940 value 0.000134
final  value 0.000087
converged
```

Neural Networks - ISOLET

```
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0.00625
>
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0
>
> b1 <- predict(a1, newdata=x.te, type='class') #, type='r
> mean(b1 != x.te$V618)
[1] 0.04583333
>
> b2 <- predict(a2, newdata=x.te, type='class') #, type='r
> mean(b2 != x.te$V618)
[1] 0.03333333
```

Neural Networks - ISOLET

```
> set.seed(123)
> a1 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0.05
# weights:  1870
...
iter 180 value 13.768831
final  value 13.768831
converged
> set.seed(456)
> a2 <- nnet(V618 ~ ., data=x.tr, size=3, decay=0.05
...
iter 230 value 13.900597
final  value 13.900596
converged
```

Neural Networks - ISOLET

```
> b1 <- predict(a1, type='class') #, type='raw')
> mean(b1 != x.tr$V618)
[1] 0
>
> b2 <- predict(a2, type='class') #, type='raw')
> mean(b2 != x.tr$V618)
[1] 0
>
> b1 <- predict(a1, newdata=x.te, type='class') #, type='r
> mean(b1 != x.te$V618)
[1] 0.01666667
>
> b2 <- predict(a2, newdata=x.te, type='class') #, type='r
> mean(b2 != x.te$V618)
[1] 0.01666667
```