

# Preventing Overfitting in Neural Networks



John Klossner, *The New Yorker*

CSC321: Intro to Machine Learning and Neural Networks, Winter 2016

# Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains noise
  - The target values may be unreliable.
  - There is **sampling error**. There will be accidental regularities just because of the particular training cases that were chosen.
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error.
  - So it fits both kinds of regularity.
  - If the model is very flexible it can model the sampling error really well. **This is a disaster.**

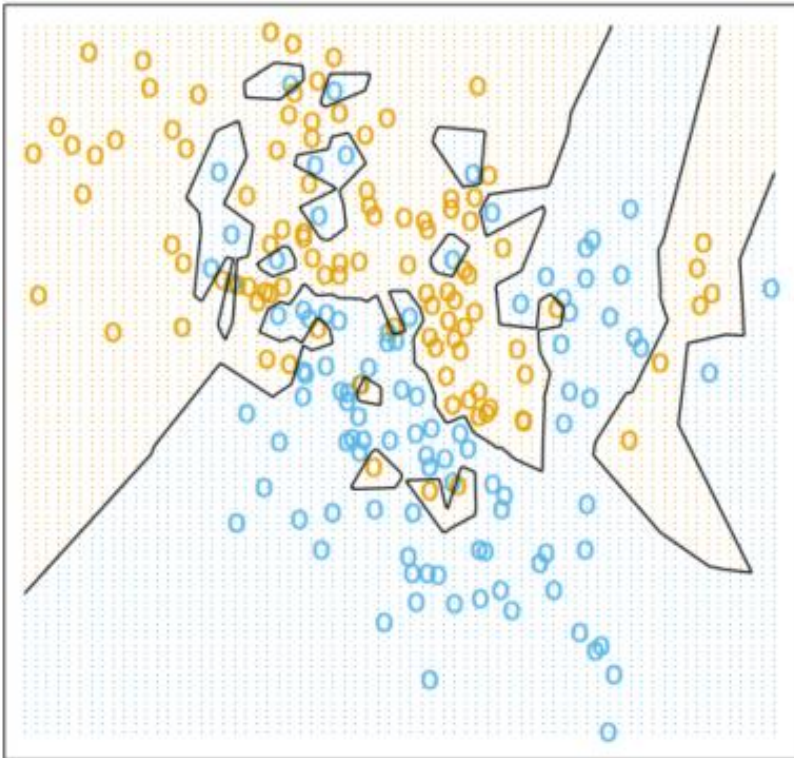
# Preventing overfitting

- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Fitting curves in 2D:
  - Only fit lines, not higher-degree polynomials (example on the board)
  - Only fit quadratics, not higher degree polynomials

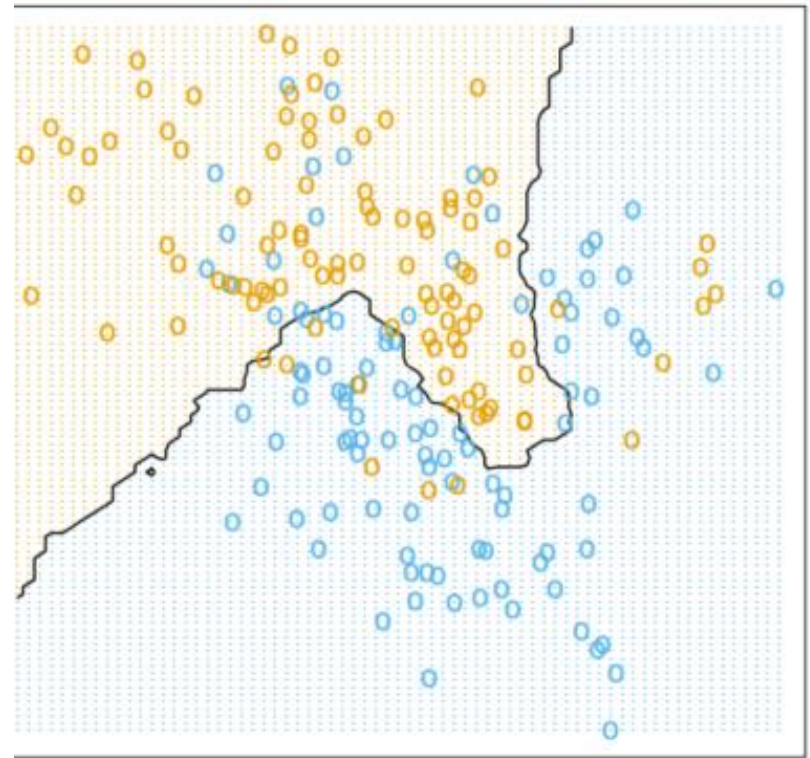
# Reminder: Nearest Neighbours

- More nearest-neighbours  $\rightarrow$  less capacity
  - More complicated decision surfaces are not possible

1-Nearest Neighbor Classifier



15-Nearest Neighbor Classifier



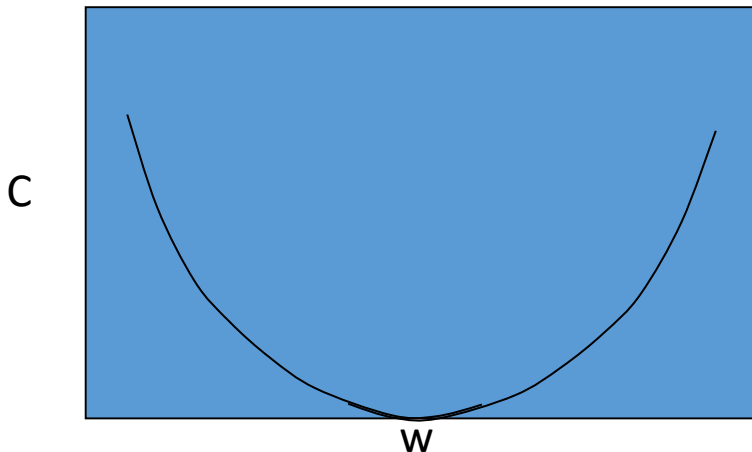
# Limiting the Capacity of a Neural Network

- Limit the number of hidden units.
- Limit the size of the weights.
- Stop the learning before it has time to overfit.

# Weight Decay: Limiting the size of the weights

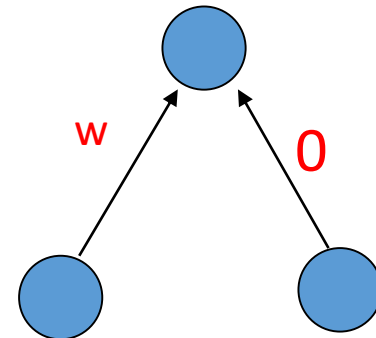
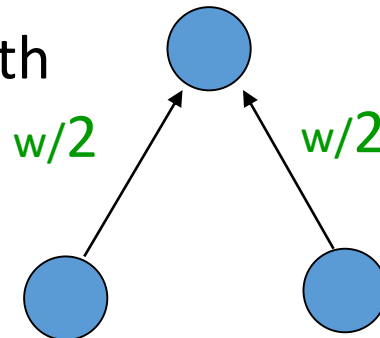
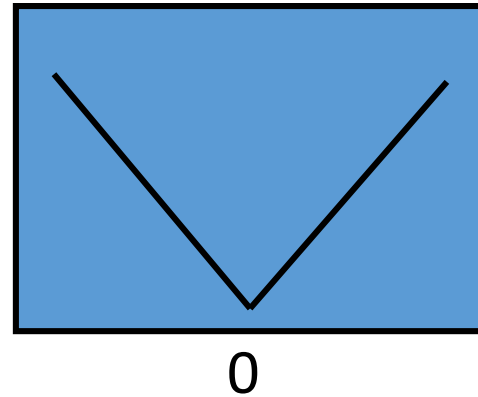
- Weight-decay involves adding an extra term to the cost function that penalizes the squared weights.
  - Keeps weights small unless they have big error derivatives.

$$cost_{WD} = cost + \frac{\lambda}{2} \sum_{i,j,k} (W^{(k,i,j)})^2$$
$$\frac{\partial cost_{WD}}{\partial W^{(k,i,j)}} = \frac{\partial cost_{WD}}{\partial W^{(k,i,j)}} + \lambda W^{(k,i,j)}$$



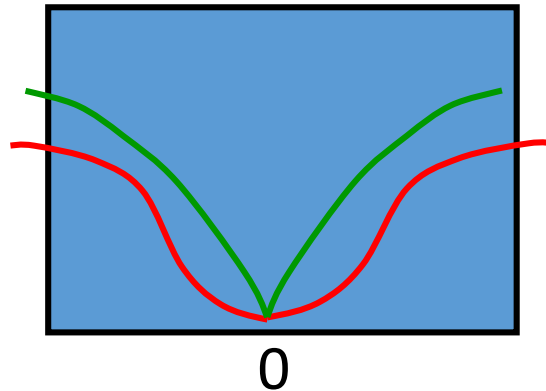
# Other kinds of weight penalty

- Sometimes it works better to penalize the absolute values of the weights. (I.e., we penalized the L1 norm of the weights rather than the L2 norm)
- Sometimes leads to smaller test errors
- Makes some weights zero
- Compared to the square penalty, which would not tend to do that
- This is sometimes helpful with interpreting the features



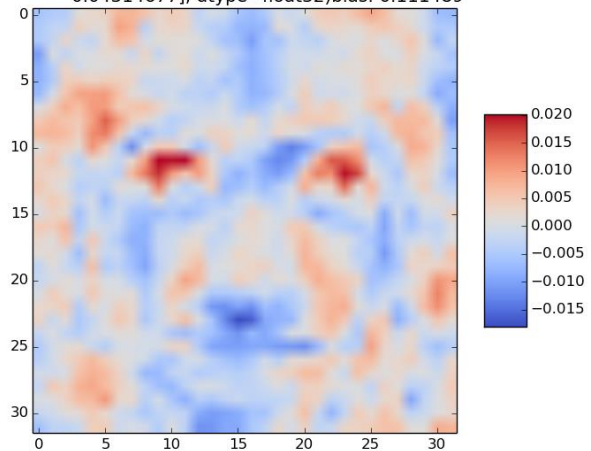
# Another kind of Weight penalty

- Sometimes it works better to use a weight penalty that has negligible effect on **large** weights.
  - Some weights *need* to be large for the neural network to work correctly!

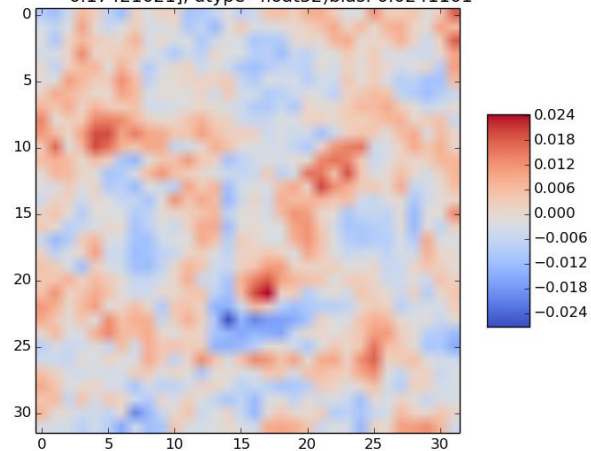




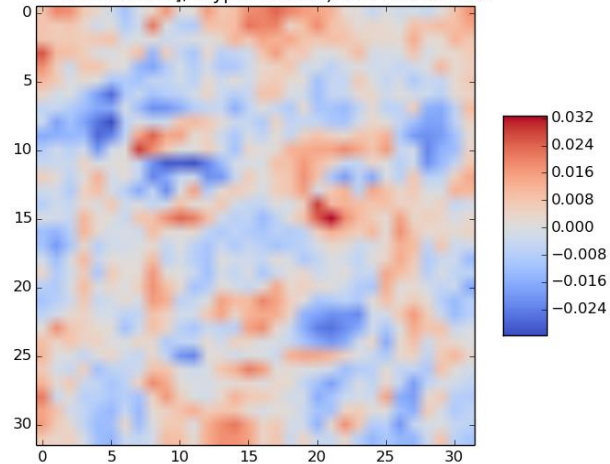
s: array([-0.1032981, -0.02623156, -0.04492124, 0.04031333, 0.09555781, 0.04314677], dtype=float32) bias: 0.111489



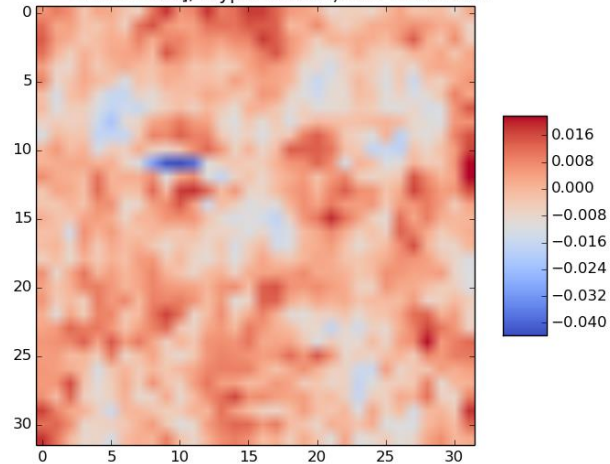
s: array([-0.05847707, 0.02304747, -0.04514949, -0.06355965, 0.02980999, 0.17421021], dtype=float32) bias: 0.0241101



s: array([0.03922304, 0.05484759, 0.06025519, 0.02333124, -0.26381665, 0.05690645], dtype=float32) bias: 0.00307018

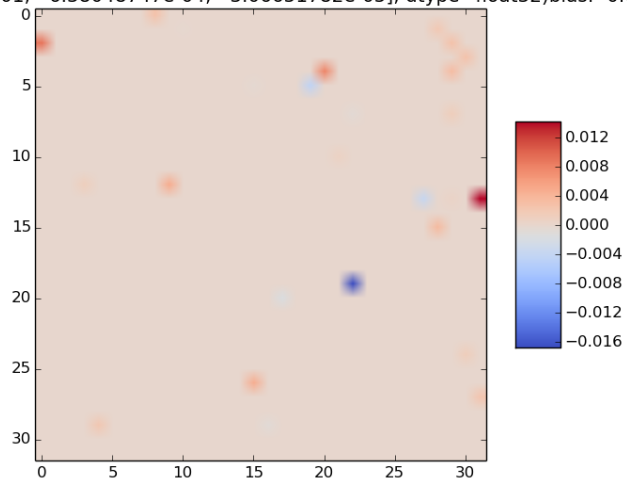


s: array([0.06559545, -0.14167207, 0.06504502, 0.01543506, -0.14153987, 0.06434423], dtype=float32) bias: 0.0287023

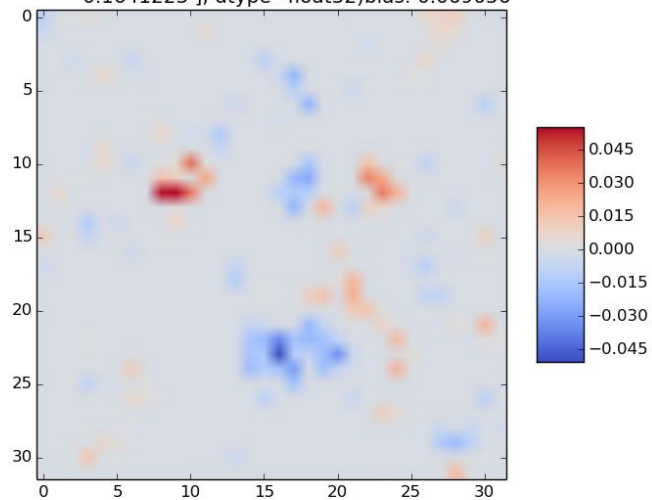


act = ['Gerard Butler', 'Daniel Radcliffe', 'Michael Vartan', 'Lorraine Bracco', 'Peri Gilpin', 'Angie Harmon']

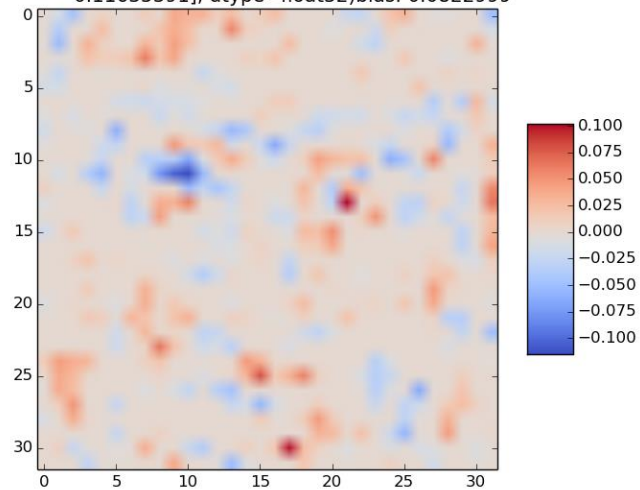
veights: array([ 3.24537978e-02, 1.03307003e-02, 1.28493230e-06,  
50160e-01, -6.38048747e-04, -3.06651782e-05], dtype=float32)bias: -0.0576:



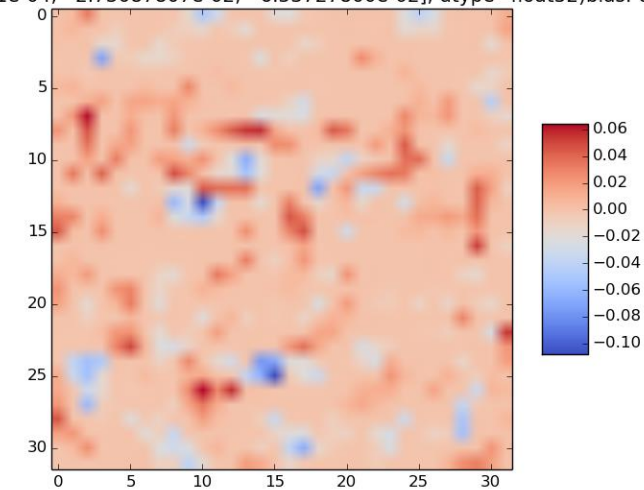
s: array([-0.29401857, -0.01724279, 0.00310232, 0.12068836, 0.0708182 ,  
0.1641223 ], dtype=float32)bias: 0.069056



s: array([ 0.22145636, -0.6399256 , 0.13758378, 0.03394366, -0.37346393,  
0.11635391], dtype=float32)bias: 0.0822999



veights: array([ -1.94610730e-01, 3.78219485e-01, -6.13273799e-01,  
55651e-04, 2.73087807e-02, -6.53727800e-02], dtype=float32)bias: 0.1243:



# The effect of weight-decay

- It prevents the network from using weights that it does not need (especially the L1 penalty).
  - This can often improve generalization a lot.
  - It helps to stop it from fitting the sampling error.
  - It makes a smoother model in which the output changes more slowly as the input changes.  $w$
- For the L2 penalty, if the network has two very similar inputs it prefers to put half the weight on each rather than all the weight on one.

# Deciding how much to restrict the capacity

- How do we decide which limit to use and how strong to make the limit?
  - If we use the test data we get an unfair prediction of the error rate we would get on new test data.
  - Suppose we compared a set of models that gave random results, the best one on a particular dataset would do better than chance. But it won't do better than chance on another test set.
- So use a separate **validation set** to do model selection.

# Using a validation set

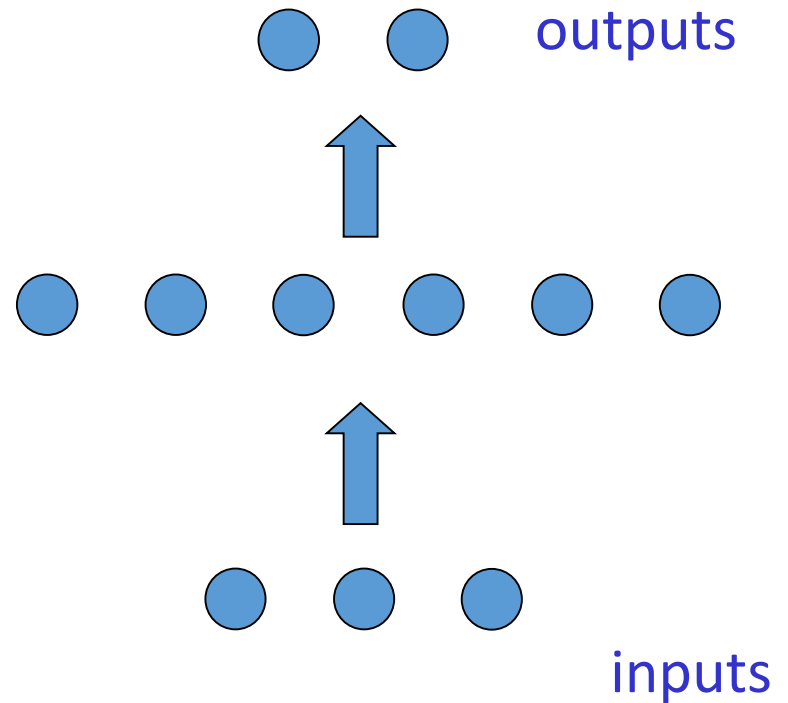
- Divide the total dataset into three subsets:
  - **Training data** is used for learning the parameters of the model.
  - **Validation data** is not used of learning but is used for deciding what type of model and what amount of regularization works best.
  - **Test data** is used to get a final, unbiased estimate of how well the network works. We expect this estimate to be worse than on the validation data.
- We could then re-divide the total dataset to get another unbiased estimate of the true error rate.

# Preventing overfitting by early stopping

- If we have lots of data and a big model, its very expensive to keep re-training it with different amounts of weight decay.
- It is much cheaper to start with very small weights and let them grow until the performance on the validation set starts getting worse (but don't get fooled by noise!)
- The capacity of the model is limited because the weights have not had time to grow big.

# Why early stopping works

- When the weights are very small, every hidden unit is in its linear range.
  - So a net with a large layer of hidden units is linear.
  - It has no more capacity than a linear net in which the inputs are directly connected to the outputs!
- As the weights grow, the hidden units start using their non-linear ranges so the capacity grows.



# Combining networks

- When the amount of training data is limited, we need to avoid overfitting.
  - Averaging the predictions of many different networks is a good way to do this.
  - It works best if the networks are as different as possible.
- If the data is really a mixture of several different “regimes” it is helpful to identify these regimes and use a separate, simple model for each regime.
  - We want to use the desired outputs to help cluster cases into regimes. Just clustering the inputs is not as efficient.



# Ways to make predictors differ

- Rely on the learning algorithm getting stuck in a different local optimum on each run.
  - A dubious hack unworthy of a true computer scientist (but definitely worth a try).
- Use lots of different kinds of models:
  - Different architectures
  - Different learning algorithms.
- Use different training data for each model:
  - **Bagging**: Resample (with replacement) from the training set: a,b,c,d,e -> a c c d d
  - **Boosting**: Fit models one at a time. Re-weight each training case by how badly it is predicted by the models already fitted.
    - This makes efficient use of computer time because it does not bother to “back-fit” models that were fitted earlier.