Lecture 9: Neural networks

Rebecka Jörnsten, Mathematical Sciences

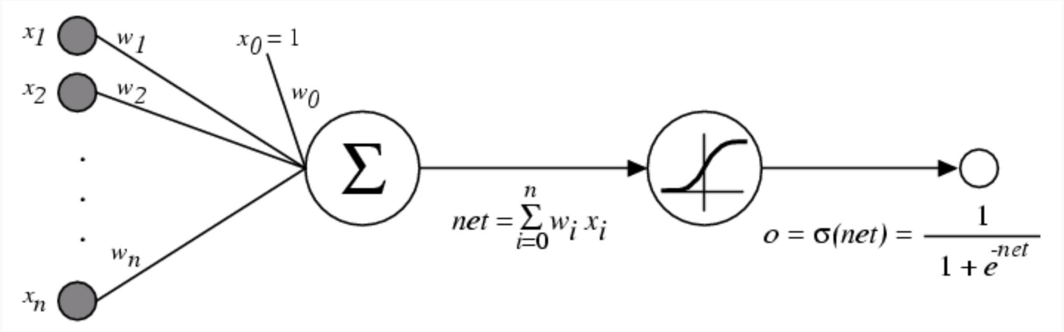
MSA220/MVE441 Statistical Learning for Big Data

7th April 2025

Revisiting logistic regression

Neural networks

- Let's first just revisit logistic regression and write it as a network type model

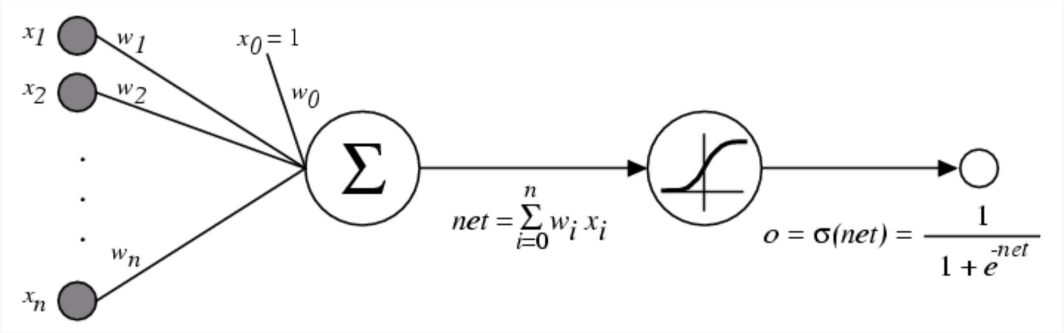


x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

credit: Artii Singh, CMU

Neural networks

- The input layer is just the features.
- The weights are our logistic regression coefficients
- At the summation node, we form the *linear predictor, z*
- We use the sigmoid function to map the linear predictor onto the interval $[0, 1]$



x_1
x_2
x_n
w_1
w_2
w_n
$x_0 = 1$
w_0
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Neural networks

- We use the logistic loss

$$-\sum_i y_i log(f_i) + (1 - y_i) log(1 - f_i)$$

- $f_i = \sigma(\sum_j \beta_j x_{ij}), z_i = \sum_j \beta_j x_{ij}$ is the linear predictor
- Take derivatives: $\frac{\partial l}{\partial \beta_j} = \frac{\partial l}{\partial f} \frac{\partial f}{\partial z} \frac{\partial z}{\partial \beta_j}$
- $\frac{\partial l_i}{\partial f_i} = -\frac{(y_i - f_i)}{f_i(1-f_i)}, \frac{\partial f_i}{\partial z_i} = f_i(1 - f_i), \frac{\partial z}{\partial \beta_j} = x_{ij}$

Putting it altogether, the sigmoid derivative $f_i(1 - f_i)$ cancels out and we obtain the simple GD update

$$\beta_j(t + 1) = \beta_j(t) + \eta \sum_i (y_i - f_i) x_{ij}$$

Neural networks

- When we have multiple classes the output layer now consists of K nodes, one for each logit/class $f_{ik}, k = 1, \cdots, K$
- $f_{ik} = \frac{e^{z_{ik}}}{\sum_l e^{z_{il}}}$ (softmax), where $z_{ik} = \sum_j \beta_j^k x_{ij}$ are the linear predictors, $\forall k$
- We use the extension of the 2-class loss: the cross-entropy

$$-\sum_i \sum_k y_{ik} log(f_{ik}), y_{ik} = 1 \; if \; obs \; i = \; class \; k$$

- Take derivatives: $\frac{\partial l}{\partial \beta_j^k} = \sum_l \frac{\partial l}{\partial f_l} \frac{\partial f_l}{\partial z_k} \frac{\partial z_k}{\partial \beta_j^k}$, where all the f_l contribute due to the "normalization" in the soft-max denominator.
- $\frac{\partial l_i}{\partial f_{li}} = -\frac{y_{il}}{f_{il}}, \frac{\partial f_{il}}{\partial z_{ik}} = -f_{il} f_{ik}, \frac{\partial f_{ik}}{\partial z_{ik}} = f_{ik}(1 - f_{ik}), \frac{\partial z_{ik}}{\partial \beta_j^k} = x_{ij}$

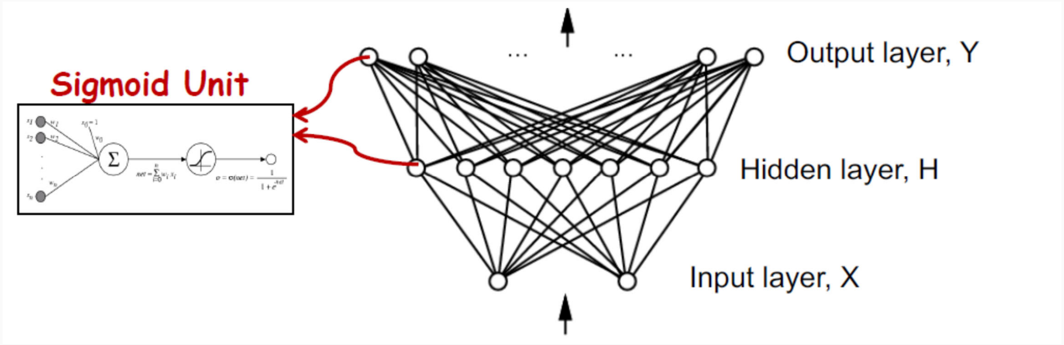
Putting it altogether, we notice that terms cancel and since $\sum_l y_{il} = 1$, we obtain the simple GD update

$$\beta_j^k(t+1) = \beta_j^k(t) + \eta \sum_i (y_{ik} - f_{ik}) x_{ij}$$

Neural networks

Neural networks

- The neural network consists of multiple layers of such logistic formulations.
- If you replace the logit function with identity (linear) and train with MSE, this is essentially PCA if the output layer is smaller than the input.
- You can of course use different nonlinear activations functions as well.


Output layer, Y
Sigmoid Unit
Hidden layer, H
Input layer, X
$x_0 = 1$
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

Neural networks

- The key will now be to use the simple chain rule formulation to obtain gradient descent updates for each layer.
- Let's refer to the nodes of the inner layers as $Z^l, l = 1, \cdots, L$ where $Z^0 = X$, the input layer, and $f(Z)$ as the output of those nodes, i.e. the sigmoid or some other nonlinear transformation (one commonly uses relu, sigmoid or tanh).
- For the case of 1 hidden layer, we can refer to the "weights", or the regression coefficients between the layers as W^1 going from X to Z^1, and W^2 going from f^1 to Z^2, with a last activation to get the network outputs f^2

Neural networks

- Let's refer to the nodes of the inner layers as $Z^l, l = 1, \cdots, L$ where $Z^0 = X$, the input layer, and $f(Z)$ as the output of those nodes, i.e. the sigmoid or some other nonlinear transformation (one commonly uses relu, sigmoid or tanh).
- For the case of 1 hidden layer, we can refer to the "weights", or the regression coefficients between the layers as W^1 going from X to Z^1, and W^2 going from f^1 to Z^2, with a last activation to get the network outputs f^2
- With the sigmoid as the nonlinear transformation between layers, following the multinomial logistic above we obtain

$$\frac{\partial L}{\partial W^2_{jk}} = -\sum_i (y_{ik} - f^2_{ik}) f^1_{ij}$$

$$\frac{\partial L}{\partial W^1_{lj}} = -\sum_i \sum_k (y_{ik} - f^2_{ik}) W^2_{jk} x_{il}$$

Neural networks

- If we want to generalize this to other activations g^1, g^2 in the two layers, we can write the updates as

$$\frac{\partial L}{\partial W^2_{jk}} \rightarrow W^2_{jk} += \eta \sum_i \underbrace{(y_{ik} - f^2_{ik})(g^2_k(W^2 f^1))'}_{\text{"error" } \delta_{ik}} f^1_{ij}$$

$$\frac{\partial L}{\partial W^1_{lj}} \rightarrow W^1_{lj} += \eta \sum_i \underbrace{\sum_k (y_{ik} - f^2_{ik})(g^2_k(W^2 f^1))' W^2_{jk} (g^1_j(W^1 X))'}_{\text{"error" } s_{ij}} x_{il}$$

  where $s_{ij} = (g^1(W^2 X)_j)' \sum_k \delta_{ik} W^2_{jk}$
- It's simply the chain rule applied backward through the network. As you can see, the update weights in the layers going backwards depend on the errors in the layer after.
- This chain of updates is called backpropagation and generalizes in the same way to multiple layers.

Neural networks - training procedure

1. Generate a set of random starting weights near 0 (initially, the network evolves as a linear model)
2. Forward propagation: using our current network parameters, pass the data through the network and obtain $f_{ik}^{L}, \forall k, i$
3. Backward propagation: Estimate the output and errors at each output layer, working backwards.
4. Perform the gradient updates
5. Iterate from 2

Neural networks - training procedure and tuning

- As with any GD procedure, the learning rate is key! If η is too large, you may get divergence instead of convergence to a local optimum
- Like with GBM, if you train for too long, you run the risk of overfitting!
- What about the network itself?
  - Wide, shallow networks can actually approximate complex functions quite well and are easier to interpret. However, you can also use a series of narrow layers to approximate your decision boundaries and build very flexible models through a series of nonlinear transformations.
  - The width and depth of your network are tuning parameters that you can use CV to select, though it's quite common to use other regularizing hyper parameters instead
  - You can use structure in the weight allocation between layers: convolutional NN, graphNN uses knowledge about the X-space, e.g. images, co-localization

Neural networks - training procedure and tuning

- Early stopping: to avoid overfitting. As with GBM, track the validation error and stop when you start overfitting
- SGD - stochastic gradient descent. Train on random batches of data. A full scan of all the data in a sequence of batches is called an epoch. Here, the smaller you make the batches, the more randomness you introduce in GD which can a) get you out of local optimal and b) prevent overfitting
- Weight decay: use the ridge or lasso penalty on the network weights
- Dropout: Prevent the network from using the output of a random set of nodes in the network during the next iteration of training. This is to avoid the network focusing the use on a subset of nodes only. (This can also be used in GBM when you drop trees to prevent the model from being too dominated by early steps in the estimation procedure).

Neural networks - training procedure and tuning

- Many more regularization methods exist
- You can penalize large $\frac{\partial f}{\partial x}$ - smoothness of the generated functions/decision boundary with respect to the data - we talked about this when we looked at Kernel Ridge Regression and nonparametric regression.
- You can penalize large $\frac{\partial f}{\partial W}$ - smoothness of the generated functions/decision boundary with respect to the parameters.
- These methods require you to sweep backward through the network twice to compute the gradient of the penalties (double backpropagation).
- You can also use more clever optimization routines than vanilla GD, e.g. using ”memory” or intertia in the training (ADAM, momentum).

Neural networks - Relation to other methods

- Lot's of current research on "opening up the black box" of neural networks
- Theory: understanding the evolution of the networks (linearization for wide networks) during training
- Connection to adaptive kernels, random feature generation,...
- I like to think of NNs as kernel logistic regression where the kernel is learnt from data.
- The sequence of nonlinear, data adaptive transformation of the inner layers of the network represents the data adaptive kernel which is then applied for classification in the last activation layer.

PCA, kPCA and Autoencoders

Comparison of Autoencoder, kPCA, and PCA

Principal Component Analysis (PCA)

- Linear dimensionality reduction technique
- Finds orthogonal directions (principal components) that explain maximum variance
- Projects data onto these components

Autoencoder

- Learns both low-dimensional representation and reconstruction of input data
- Can capture non-linear relationships and complex structures in data
- May overfit or underfit the data if not properly regularized
- Requires tuning of architecture, activation functions, and regularization techniques

Comparison of Autoencoder, kPCA, and PCA

Kernel Principal Component Analysis (kPCA)

- Non-linear extension of PCA using kernel functions
- Captures complex patterns and non-linear relationships in data
- Can handle non-linear dimensionality reduction and feature extraction
- Requires selection of the kernel and tuning of hyperparameters

Considerations

- Autoencoders and kPCA can capture non-linear patterns, while PCA is limited to linear relationships
- Autoencoders are powerful but require more computational resources and hyperparameter tuning
- kPCA and PCA are efficient and widely used, but may not capture complex non-linear patterns as effectively as autoencoders

Neural Network Classifier as an Autoencoder with a Classification Layer

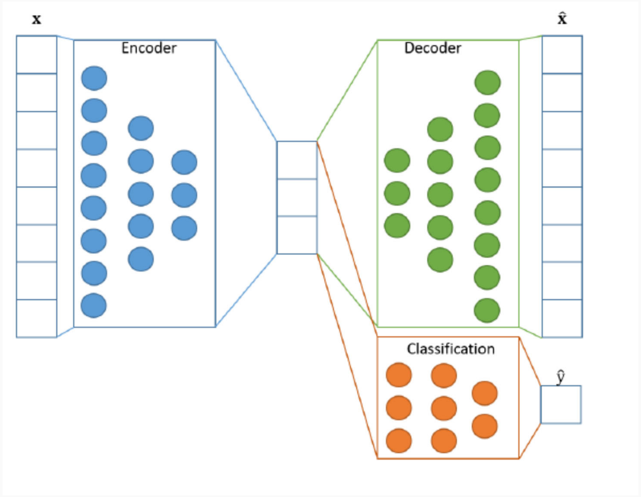
Autoencoder

- Unsupervised learning neural network architecture
- Consists of an encoder and a decoder
- Learns to reconstruct input data at the output layer
- Encodes the input data into a lower-dimensional representation at the bottleneck layer

AE based Neural Network Classifier

- Extends the autoencoder by adding a classification layer on top of the encoder
- Uses the learned representation from the autoencoder for classification
- The classification layer maps the encoded features to class labels
- Enables both unsupervised feature learning and supervised classification

Neural Network Classifier as an Autoencoder with Classification Layer



Neural Network Classifier as an Autoencoder with Classification Layer

Benefits of Autoencoder-based Classification

- Dimensionality reduction: The encoder reduces the input data to a lower-dimensional representation.
- Joint learning: The autoencoder and classification layer are trained together, leveraging the power of both unsupervised and supervised learning.
- It is similar to kernel-based methods we have discussed with the added flexibility of the kernel being data adaptive.

Considerations

- Network architecture: The design and depth of the autoencoder and classification layers impact the model's performance.
- Hyperparameter tuning: Proper selection of activation functions, learning rate, regularization, etc., is crucial.

NN, KRR and SVMs

Similarities

- Can handle nonlinear decision boundaries through kernels (fixed/data driven)
- Comparable performance for many problems though NNs have an edge for complex problems if enough data and computational resources are available

NN, KRR and SVMs

Differences

- losses (log, hinge, cross-entropy)
- computational burden for SVMs and KRR vastly lower than for NN
- SVMs more robust wrt mislabeling than KRR and NNs BUT you can of course regularize these methods to adress the challenge of noisy labels.
- NNs are much more data hungry!
- NNs are sensitive to initializations
- Many more parameters and hyperparameters to train for NNs

Summary: Don't underestimate the value of classical kernel-based ML methods.