COMP 652: Machine Learning

Lecture 6

Today

- Recall: Backprop algorithm
- Various tricks and tips about backprop & ANN training
  - Weight initialization
  - Input/output representations
  - Momentum
  - Learning rates; delta-bar-delta
  - Second order methods
- Network architecture
  - Overfitting & overtraining issues
  - “Desctructive” methods for avoiding overfitting

Recall: Backprop (for 3 layer nets, in particular)

- Consider a three-layer network with N inputs, H sigmoidal hidden units, and 1 sigmoidal output unit.
- For a single training instance $\langle \mathbf{x_i}, y_i \rangle$, the sum-squared error is $J_i = \frac{1}{2}(y_i - o_{N+H+1})^2$, where o_{N+H+1} is the output of the network upon input $\mathbf{x_i}$.
- To compute $\nabla_{\mathbf{w}} J_i$:

  1. Use a feedforward pass to compute the outputs of all units.
  2. Compute the δ's at every unit:

$$\begin{aligned} \delta_{N+H+1} &= (y_i - o_{N+H+1}) o_{N+H+1} (1 - o_{N+H+1}) \\ \delta_l &= \delta_{N+H+1} w_{N+H+1,l} o_l (1 - o_l) \text{ for hidden unit } l \end{aligned}$$

  3. Update the weights: $w_{i,j} \leftarrow w_{i,j} + \alpha_{i,j} \delta_i o_j$, where $\alpha_{i,j}$ is a learning rate (or step-size).

- For multiple output units or hidden layers, see previous lecture slides for more general treatment.

Backpropagation variations

- The previous slide tells us how to compute $\nabla_{\mathbf{w}} J_i$.
- This can be used in an to incremental (or stochastic) gradient descent
  - We could loop through all training instances, taking a single gradient descent step on each. Such a loop is called an *epoch*.
  - We could repeatedly chose a training instance at random, and take a step on that instance.
- We can also do a true gradient descent. How?

Backpropagation variations

- The previous slide tells us how to compute $\nabla_{\mathbf{w}} J_i$.
- This can be used in an to incremental (or stochastic) gradient descent
  - We could loop through all training instances, taking a single gradient descent step on each. Such a loop is called an *epoch*.
  - We could repeatedly chose a training instance at random, and take a step on that instance.
- We can also do a true gradient descent.

$$J = \sum_{i=1}^{m} J_i \qquad \Rightarrow \qquad \nabla_{\mathbf{w}} J = \sum_{i=1}^{m} \nabla_{\mathbf{w}} J_i$$

- Algorithm can be easily generalized to predict probabilities, instead of minimizing sum-squared error.
- For other feedforward architectures, gradient can be computed similarly.

Convergence of backpropagation

- Backpropagation performs gradient descent over ***all*** the parameters in the network
- Hence, if the learning rate is appropriate, the algorithm is guaranteed to converge to a *local minimum* of the cost function
  - NOT the global minimum
  - Can be much worse than global minimum
  - There can be MANY local minima (Auer et al, 1997)
- Partial solution: **restarting** = train multiple nets with different initial weights.
- In practice, quite often the solution found is very good
- . . . but other tricks and tips can help a lot, too.

Initializing the weights

- As with any local search method, the starting point of the search can have a large impact on the outcome of the search.
- DO NOT initialize all the weights to zero! Why?

Initializing the weights

- As with any local search method, the starting point of the search can have a large impact on the outcome of the search.
- DO NOT initialize all the weights to zero! Why?
  $\Rightarrow$ All hidden units, for example, will behave identically, and the ANN will be as if it had only one hidden unit. You need to break symmetry somehow.
- If initial weights are too large, unit outputs will likely be saturated — outputting near zero or near one. The SSQ error surface is very flat in such cases.
- Typically, weights are initialized to small, random numbers.
  (E.g., normal with standad deviation 0.1, or uniform in the interval $[-0.1, +0.1]$.)

Choosing a good input encoding: Discrete inputs

- Discrete inputs with k possible values are often encoded using a "1 hot" or "1-of-k" encoding:
  - k input bits are associated with the variable (one for each possible value)
  - For any instance, all bits are 0 except the one corresponding to the value found in the data, which is set to 1
  - If the value is missing, all inputs are set to 0
- This particularly makes sense of the values are unrelated. (E.g., schools McGill, UdeM, Toronto, ...)
- What if the values are ordered? (E.g., grades A, B, C, ...)

Choosing a good input encoding: Discrete inputs

- Discrete inputs with k possible values are often encoded using a "1 hot" or "1-of-k" encoding:
  - k input bits are associated with the variable (one for each possible value)
  - For any instance, all bits are 0 except the one corresponding to the value found in the data, which is set to 1
  - If the value is missing, all inputs are set to 0
- This particularly makes sense of the values are unrelated. (E.g., schools McGill, UdeM, Toronto, . . .)
- What if the values are ordered? (E.g., grades A, B, C, . . .)
  - 1-hot encoded loses the order information
  - Thermometer encoding – turning on all bits $\leq$ the input value – retains it.

Choosing a good input encoding: Real-valued inputs

- For continuous (numeric) inputs, it is important to *scale* the inputs so they are all in a common, reasonable range
- One standard transformation is to "normalize" the data, i.e., make it such that it has mean 0, unit variance, by subtracting the mean and dividing by the standard deviation
- When is this good? When is this bad?

- Alternative representations:
  - 1-hot encoding - what is the disadvantage?

  - Thermometer encoding

Choosing a good input encoding: Real-valued inputs

- For continuous (numeric) inputs, it is important to *scale* the inputs so they are all in a common, reasonable range
- One standard transformation is to "normalize" the data, i.e., make it such that it has mean 0, unit variance, by subtracting the mean and dividing by the standard deviation.
- When is this good? When is this bad? Good if the data is roughly normal, but bad if we have outliers
- Alternative representations:
  - 1-hot encoding - what is the disadvantage? Potentially lots of values! Also, the ordering of values is lost
  - Thermometer encoding

Output

- A network can have several output units
- This can be useful when we want to predict a real-valued variable, and it has several ranges
  (E.g., Dean Pomerleau's autonomous driving car)
- Also useful when predicting the probabilities of more than two possible classes/outcomes
  (E.g., TD-gammon)
- Alternatively, we can allow one output unit, without a sigmoid function
- Normalization can be used if the output data is roughly normal

Adding momentum

On the t-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \alpha_{ij} \delta_j x_{ij}$$

we do:

$$\Delta w_{ij}(t) \leftarrow \alpha_{ij} \delta_j x_{ij} + \beta \Delta w_{ij}(t-1) \text{ where } \beta \in (0,1)$$

The second term is called **momentum**.

Adding momentum

On the t-th training sample, instead of the update:

$$\Delta w_{ij} \leftarrow \alpha_{ij}\delta_j x_{ij}$$

we do:

$$\Delta w_{ij}(t) \leftarrow \alpha_{ij}\delta_j x_{ij} + \beta\Delta w_{ij}(t-1) \text{ where } \beta \in (0,1)$$

The second term is called **momentum**.

Advantages:

- Easy to pass small local minima
- Keeps the weights moving in areas where the error is flat
- Increases the speed where the gradient stays unchanged

Disadvantages:

- With too much momentum, it can get out of a global maximum!
- One more parameter to tune, and more chances of divergence

$\Rightarrow$ There are many variants!

Choosing the learning rate

- Backprop is VERY sensitive to the choice of learning rate
  - Too large $\Rightarrow$ divergence
  - Too small $\Rightarrow$ VERY slow learning
  - The learning rate also influences the ability to escape local optima
- Sometimes, different learning rates are used for units inn different layers
  - Particularly early in training, the partial derivs for the input-to-hidden layer weights are much smaller than the hidden-to-output layer weights.

Adjusting the learning rate: Delta-bar-delta

- Heuristic method, works best in batch mode (though there have been attempts to make it incremental)
- The intuition:
  - If the gradient direction is stable, the learning rate should be increased
  - If the gradient flips to the opposite direction the learning rate should be decreased
- A running average of the gradient and a separate learning rate is kept for each weight
- If the new gradient and the old average have the same sign, increase the learning rate by a constant amount
- If they have opposite sign, decay the learning rate exponentially

Delta-bar-delta more formally

- After t training epochs, let $g_{i,j}(t) = \frac{\partial}{\partial w_{i,j}} J$.
- For some $0 < \beta < 1$, consider the recency-weighted average of the partial derivatives:

$$\begin{aligned} \bar{g}_{i,j}(0) &= 0 \\ \bar{g}_{i,j}(t) &= (1-\beta) g_{i,j}(t-1) + \beta \bar{g}_{i,j}(t-1) \text{ for } t > 0 \end{aligned}$$

- The learning rate for weight $w_{i,j}$ at epoch $t+1$ is then:

$$\alpha_{i,j}(t+1) = \begin{cases} \alpha_{i,j}(t) + \kappa & \text{if } \bar{g}_{i,j}(t) \text{ and } g_{i,j}(t) \text{ of same sign} \\ (1-\gamma)\alpha_{i,j}(t) & \text{if } \bar{g}_{i,j}(t) \text{ and } g_{i,j}(t) \text{ of opposite signs} \\ \alpha_{i,j}(t) & \text{otherwise} \end{cases}$$

A brief taste of second-order methods

- Recall Newton's method for finding the zero of a function $g : \Re \rightarrow \Re$
- At point u_i, approximate the function by a straight line (its tangent)
- Solve the linear equation for where the tangent equals 0, and move the parameter to this point:

$$u_{i+1} = u_i - \frac{g(u_i)}{g'(u_i)}$$

Application to machine learning

- We want to optimize an error function f, so we can apply Newton's method to find the zeros of f'
- We obtain the iteration:

$$u_{i+1} = u_i - \frac{f'(u_i)}{f''(u_i)}$$

- Note that there is no step size parameter here!
- This is a ***second-order*** method, because it requires computing the second derivative
- But, if our error function is quadratic, this will find the global optimum in one step!

Second-order methods: Multivariate setting

- If we have an error function J that depends on many variables, we can compute the Hessian matrix, which contains the second-order derivatives of J:

$$H_{ij} = \frac{\partial^2 J}{\partial w_i \partial w_j}$$

- The inverse of the Hessian gives the "optimal" learning rates
- The weights are updated as:

$$\mathbf{w} \leftarrow \mathbf{w} - H^{-1} \nabla_{\mathbf{w}} J_{\mathbf{w}}$$

- This is also called Newton-Raphson method

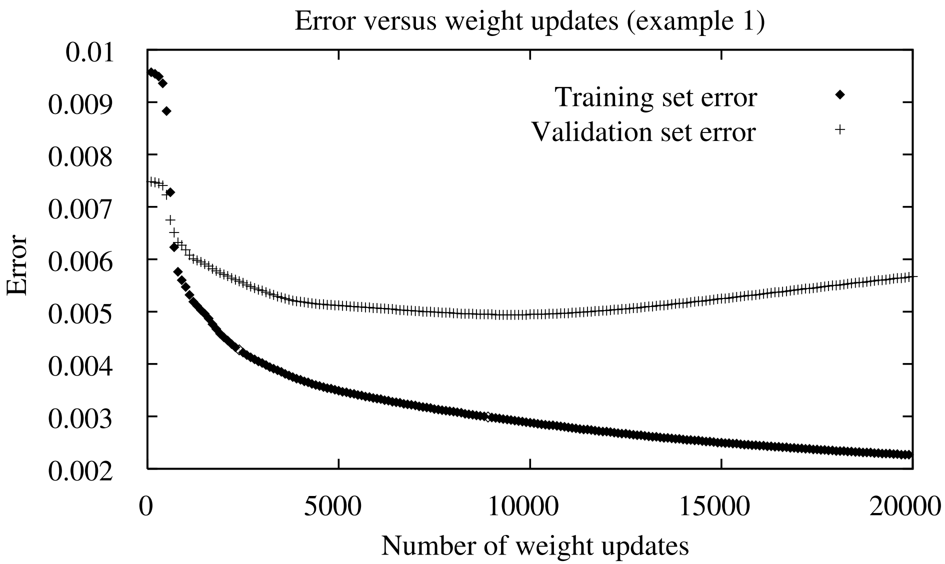
Gradient versus second-order methods

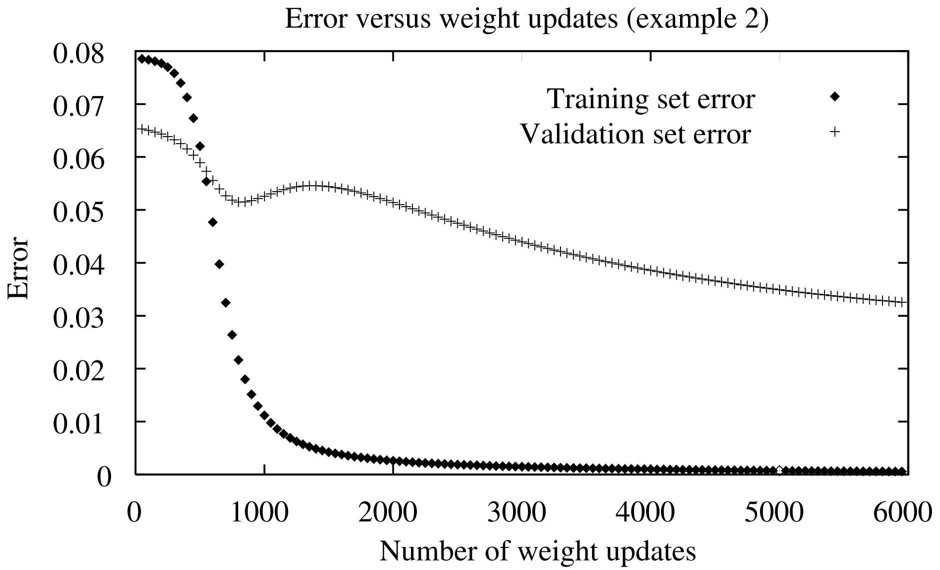
- Newton's method:
  - Usually requires significantly fewer iterations than gradient descent; especially good when already close to minimum.
  - But each iteration more computationally expensive, since it requires computing and inverting the Hessian (though their are cheaper approximations)
  - Used only in batch mode, not incremental.
  - Can diverge!
- Gradient descent (backprop):
  - Is computationally cheaper (per epoch)
  - Convergence to local minimum can be guaranteed (for standard backprop)
  - Training slows as approaches minimum
  - Momentum and Delta-bar-delta are computationally cheap, and commonly used to speed learning.

How large should the network be?

- Overfitting occurs if there are too many parameters compared to the amount of data available
- Choosing the number of hidden units:
  - Too few hidden units do not allow the concept to be learned
  - Too many lead to slow learning and overfitting
  - If the n inputs are binary, $\log n$ is a good heuristic choice
- Choosing the number of layers
  - Always start with one hidden layer
  - Never go beyond 2 hidden layers, unless the task structure suggests something different

Overtraining in feed-forward networks





- This is a different form of overfitting, which occurs when weights take on large magnitudes, pushing the sigmoids into saturation.
- Effectively, as learning progresses, the network has *more parameters*.
- *Use a validation set to decide when to stop training!*

k-fold cross-validation

1. Split the training data into k partitions (folds)
2. Repeat k times:
   (a) Take one fold to be the test set
   (b) Take one fold to be the validation set
   (c) Take the remaining $k-2$ folds to form the training set
   (d) We train the parameters on the training set, using the validation set to decide when to stop, then measure $J_{train}(i)$ and $J_{test}(i)$ on fold i
3. Report the average of $J_{train}(i)$ ***and*** the average of $J_{test}(i)$, $i = 1, \ldots k$.

Magic number: $k = 10$.

More on cross-validation

- It is good to ensure the same distribution of examples in each fold
- If two algorithms are compared, it should be *on the same folds*
- We get an idea not only of the average performance, but also of the variability in the algorithm
- If there is too little data, or you want a better idea of what the "hard" examples are, use leave-one-out cross-validation

Finding the right network structure

- *Destructive methods* start with a large network and then remove (prune) connections
- *Constructive methods* start with a small network (e.g. 1 hidden unit) and add units as required to reduce error

Some destructive methods

- Simple solution: consider removing each weight in turn (by setting it to 0), and examine the effect on the error
- Weight decay: give each weight a chance to go to 0, unless it is needed to decrease error:

$$\Delta w_j = -\alpha_j \frac{\partial J}{\partial w_j} - \lambda w_j$$

  where λ is a decay rate
- Optimal brain damage:
  - Train the network to a local optimum
  - Approximate the saliency of each link or unit (i.e., its impact on the performance of the network), using the Hessian matrix
  - Greedily prune the element with the lowest saliency

  This loop is repeated until the error starts to deteriorate