New York University: Machine Learning in Economics

Lecture 8: Neural Networks

Dr. George Lentzas

Introduction

What are Neural Networks?

- Neural Networks (NNs) are statistical models that extract combinations of the predictors as new "derived features" and then model the dependent variable as a function of these derived features.
- The term NN encapsulates a vast collection of different models and was originally motivated (along with a lot of the terminology of the NN literature) by modeling the human brain.
- The goal of all NNs is to approximate some function $f^*(\mathbf{x})$ via $f(\mathbf{x};\boldsymbol{\theta})$ and learn the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.
- We will start with the most common model, the single hidden layer network (a.k.a. Single Layer Perceptron). This is illustrated in the graph below.

Introduction

Single Layer Perceptron

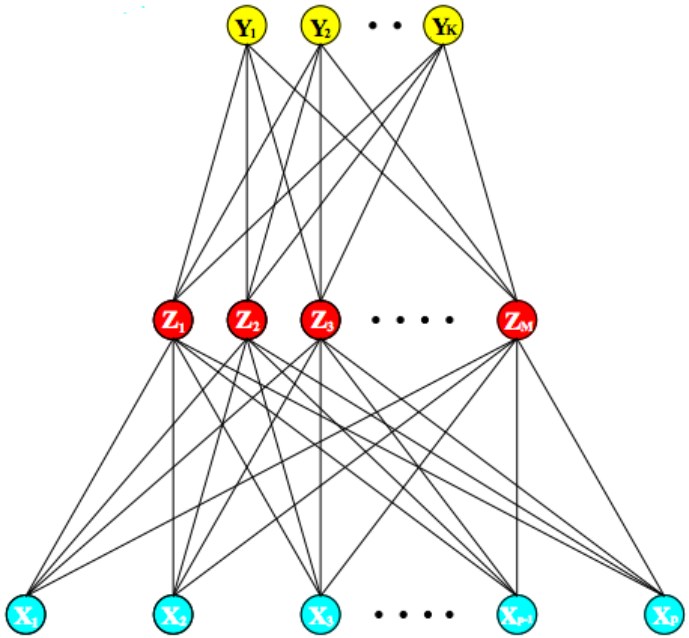


Figure: A single hidden layer, feed-forward, neural network.

SLP Theory

Definitions

- A SLP is a two stage model that can be used for both classification *or* regression.
- For K-class classification there are K output units (as illustrated above) with the k^{th} unit modeling the probability of obtaining class k.
- For regression we usually have $K = 1$ although this is not really a restriction.
- The derived features Z_m are generated using linear combinations of the predictors and are called the *hidden units* because they are not directly observed.
- Then the output variable Y_k is modeled as a function of linear combinations of the derived Z_m hidden units.

SLP Theory

1. Z=take input vars (X) through a logistic (sigmoidal) function
2. T=take hidden vars Z and construct linear combination
3. Y = T if regression and Y=g(T) if classification, g(T) is the softmax fun

The Traditional SLP Model

- This can be summarized as follows:

$$Z_m = \sigma\left(\alpha_{0m} + \alpha_m' \mathbf{X}\right), m = 1, \ldots, M$$

this has intercept & slope

$$T_k = \beta_{0k} + \beta_k' \mathbf{Z}, k = 1, \ldots, K$$

$$f_k(X) = g_k(\mathbf{T}), k = 1, \ldots, K.$$

- Here $\sigma(x)$ is the sigmoid function

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

- The output function $g_k(T)$ is the identity function $g_k(T_k) = T_k$ for regression and the softmax function

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

for classification.

nice function since takes a real vector as input and

SLP Theory

The Model

- We can see the sigmoid function below. This function lies at the heart of the traditional SLP as it allows us to generalize the simple linear model.

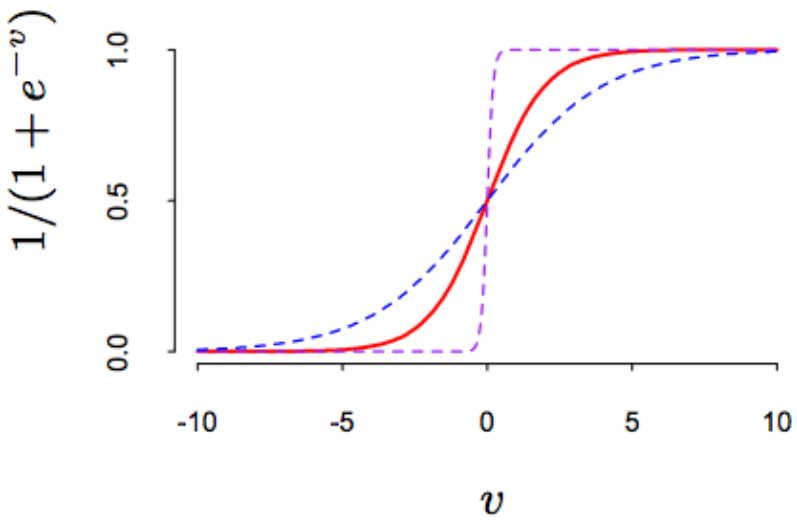


Figure: A plot of the sigmoid function $\sigma(s \times u)$. The parameter s is a scaling parameter tha controls the activation right, with smaller s causing a smoother activation.

SLP Practice

m: number of features
K number of hidden units

Fitting Neural Networks

- In order to fit a SLP to the data we need to estimate the following set of parameters, denoted by θ :

intercept, slope params

$$\left(\alpha_{0m}, \boldsymbol{\alpha_m}\right), m = 1, \ldots, M$$ input to hidden layer params

$$\left(\beta_{0k}, \boldsymbol{\beta_k}\right), k = 1, \ldots, K$$ input to hidden layer params

for a total of $M(p+1)$ and $K(M+1)$ parameters respectively.

- For regression it is common to minimize the sum of squared errors and for classification we minimize the squared error or cross entropy (deviance).

SLP Practice

Overfitting

- Typically SLPs are over-parameterized and will likely overfit the data; hence we typically do not want to search for a global minimum.
- One approach is to put a stop to the number of iterations in the optimization routine. This can be either a predetermined stop (e.g. 100 iterations) or based on validation (stop once the validation error starts to increase).
- Another approach is to use a penalty term, similar to ridge regression. This involves adding a penalty term to the error function to be minimized

$$R(\theta) + \lambda J(\theta)$$

where $R(\theta)$ is the original minimization criterion (e.g. squared errors) and

$$J(\theta) = \sum_{km} \beta_{km}^2 + \sum_{ml} \alpha_{ml}^2$$

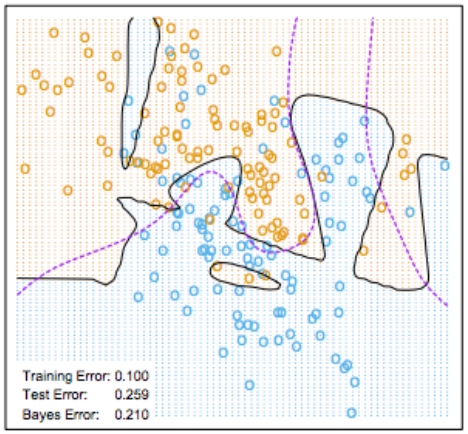
and λ a tuning parameter to be estimated by cross-validation.

- The following graph shows an example of the perils of overfitting and weight decay from ESL. The data is from the mixture example of ESL Chapter 2.

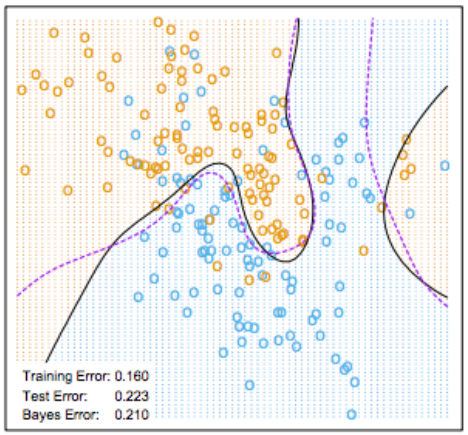
Discussion

Overfitting

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02



Deep Forward Networks

Introduction

- Deep Forward Networks (also called Multilayer Perceptrons, or MLPs) are the natural generalization of SLPs to allow for multiple hidden layers. They are the quintessential deep learning model.
- To understand why it makes sense to have multiple hidden layers one needs to appreciate that NNs are non-linear function approximation tools that aim at achieving statistical generalization.
- In order to extend linear models to represent nonlinear functions of $\mathbf{x}$ we can apply a linear model to a transformed input $\phi(\mathbf{x})$ where ϕ is a non-linear transformation.
- The question is how do we choose the mapping ϕ?

Deep Forward Networks

Introduction

- One option is to choose a generic ϕ (e.g. a kernel) based on the principle of local smoothness. The problem with this is that it does not introduce enough prior information to solve advanced generalization problems.
- Another option is to manually engineer ϕ. This works well in some domains where we have intuition about the features but not always. Also it allows little transfer of knowledge between domains.
- The answer of MLPs is to learn ϕ by using a model

$$y = f(\mathbf{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\mathbf{x}; \boldsymbol{\theta})' \boldsymbol{w}$$

  This is highly generic by choosing a broad family of functions but can generalize well.
- The catch is that the optimization problem becomes non-convex.

Deep Forward Networks

Design Decisions

- Designing MLPs involves multiple key decisions regarding:
  1. optimization algorithms
  2. choice of cost function
  3. network architecture
  4. generalization
- Using MLPs in practice is both science and art as the practical considerations can easily get overwhelming.

Introduction

What about Optimization?

- Optimization in the context of deep learning involves finding parameters $\boldsymbol{\theta}$ of a neural network that reduce a cost function $J(\boldsymbol{\theta})$ which usually includes a performance measure evaluated on the entire training set as well as additional regularization terms.
- Ideally we would like to minimize

$$E^*(\mathbf{w}) = \mathrm{E}_{(\mathbf{x},y)\sim p_{data}} L((\mathbf{x};\mathbf{w}), y)$$

but of course this quantity (sometimes referred to as 'risk') is unknown.

- Instead we can minimize the sample version of this,

$$E(\mathbf{w}) = \mathrm{E}_{(\mathbf{x},y)\sim \hat{p}_{data}} L((\mathbf{x};\mathbf{w}), y) = \frac{1}{m}\sum_{i=1}^{m} L((\mathbf{x_i};\mathbf{w}), y_i)$$

known as empirical risk. This unfortunately would lead to overfitting so in practice the quantity we optimize is even more different than the quantity we truly want to optimize (risk).

Optimization Algorithms

Error Surfaces

- NNs, due to the associated non-convexity of the cost function, are trained by iterative, gradient based optimizers that find a numerical local minimum.
- Learning in a neural network is formulated in terms of a minimization of an error function $E(\mathbf{w})$ (for example sum of squared residuals) which is generally assumed to be a continuous and differentiable function of the weight vector $\mathbf{w}$.
- There is a number of algorithms that deal with this problem, some of which we will review below.
- The error function E will typically be highly non-linear and complex so closed form solutions will typically be non-attainable. In addition we expect that there will be multiple local minima to which the optimization algorithm might converge to, instead of the global minimum.

Optimization Algorithms

Error Surfaces

- The general format of the optimization algorithm is of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)}$$

  where τ denotes the iteration step and different algorithms imply different choices for the weight vector increment $\Delta\mathbf{w}^{(\tau)}$.
- Neural Networks exhibit a high degree of symmetry (for example a two layer network has a symmetry factor of $M!2^M$. Thus for any weight solution there will be $M!2^M$ equivalent solutions which generate an identical network mapping and thus the same exact value for the error function.
- This has the profound implication that any local or global minima will be replicated a large number of times throughout the weight space.

Optimization Algorithms

Practical Considerations

- Most optimization algorithms involve setting initial weights as starting points for the algorithm. Random values are typically used to avoid instability from choosing symmetric starting weights.
- Initial values are also usually small to avoid pushing the derivative of the sigmoid activation function to be driven to the edges where the gradient of the error would also become small (flat error surface) early on. With near-zero weights the model starts as roughly linear and becomes non-linear as the weights increase.
- On the other hand, too small (or zero) starting weights reduce the model to approximate linearity so a common choice is that the summed inputs to the sigmoidal function be of order unity.
- It is best to standardize all inputs to have zero mean and unit variance. This not only ensures all inputs are treated equally in the optimization routine but also makes choosing a meaningful range for the starting weights easier.
- In the case of time series it is also important that the data represent a stable functional relationship to be estimated, so deterministic and stochastic trend should be removed (by de-trending and/or differencing).

Optimization Algorithms

Gradient Descent

- Gradient (steepest) Descent is one of the most intuitive optimization algorithms.
- The algorithm starts at some initial guess of the weights and at each iteration moves a short distance in the direction of the greatest rate of decrease of the error (i.e. in the direction of the negative gradient evaluated at $\mathbf{w}^{(\tau)}$) so that

$$\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E|_{\mathbf{w}^{(\tau)}}$$

- The parameter η is called the learning rate and must be chosen by the researcher.
- In classical Gradient Descent the gradient is evaluated using the entire sample
- Simple gradient descent is unfortunately very inefficient since it requires a large number of iterations particularly in areas where the gradient oscillates strongly.

to calculate this error we need to calculate the sum of (y_i - y_hat)^2 for all i. if you have a large dataset thats a lot of calculations in only one optimization step

Deep Neural Networks

Stochastic Gradient Descent & Mini Batches

- **Stochastic Gradient Descent (SGD)** is an estimation simplification trick that speeds up the time needed to train our DNN. SGD builds on Gradient Descent: the idea is that instead of estimating the gradient exactly using the entire sample, at each iteration we estimate it using one randomly chosen sample point. This introduces noise but makes the training much faster.

this selets only a random batch of the sample to calcylate the error

$$\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E|_{\mathbf{w}^{(\tau)}} (\mathbf{x}_t)$$

where $\mathbf{x}_t$ is a randomly picked data point and η is the learning rate.

- This approach has the advantage of speed as well as a stochastic nature which might allow it to escape from local minima. However using only one observation might make the estimate of the gradient very noisy and unreliable.

Mini Batching

- A middle ground is to use **mini-batching**. This involves computing the gradient using a small "batch" (a mini sample). Typically numbers between $10 - 50$ have been used. This is the approach used in almost all modern applications.

Optimization Algorithms

Adding Momentum

- **Momentum** is related technique that tries to deal with the issue of noisy ("oscillatory") SGD and to speed up learning in long and flat "ravines" of the optimization function. The idea is to update the weight using a combination of current and previous updates, effectively adding an inertia term to the equation above.
- SGD with Momentum adds an inertia term to smooth out such oscillations and increase the effective learning rate in oscillatory gradient areas. This is done by

$$\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E|_{\mathbf{w}^{(\tau)}} \left(\{\mathbf{x}_t^{batch}\} \right) + \mu \Delta \mathbf{w}^{(\tau-1)}$$

- In practice this leads to a significant improvement in the performance of SGD but at the cost of adding a second parameter (the momentum parameter μ) which needs to also be chosen.

Optimization

Challenges

- For many non-convex function local minima are rare compared to other points of zero gradient: saddles points. For first order optimization algorithms this might pose a serious issue as the gradient can become very small in saddle points. For Newton's method type algorithms, saddle points are a serious problem which is why (given the proliferation of saddle point in neural networks) such algorithms are not commonly used.
- Deep neural networks often have very steep regions resembling multi-dimensional cliffs, the result of multiplication of several large weights together. This poses a problem for GD and its variants as it usually jumps off the cliff structure altogether.
- A good way to avoid this issue is a technique called gradient clipping which reduces the GD step using some pre-determined methodology to avoid cliff jumps.

Optimization

Challenges

- So far we have been mostly focused on local properties of the loss functions and associated issues for the commonly used optimization algorithms.
- A different set of problem arises from a more global problem, namely poor correspondence between the local and global structures of the loss function. This happens when the direction that results in the most improvement locally does not point toward distant regions of much lower cost. gradient at local points do not point to global min
- Ultimately the question is if there is a low cost local minimum that can be reached by a path that local descent can follow. In high dimensions it is possible that this is usually the case but in practice it also depends on choosing good initial points for optimization algorithm to use.

Optimization

State of the Art Algorithms

- One of the most difficult hyper-parameters to tune is the learning rate of GD. Perhaps we can use a different learning rate for each parameter and automatically adapt these learning rates throughout the course of learning.
- Recently, a number of mini-batch GD methods have been introduced trying to adapt the learning rates dynamically.
- AdaGrad: adapts the learning rates by scaling them inversely proportional tot he square root of the sum of all historical squared values of the gradient. This performs well in some but not all deep neural networks due to the accumulation of memory from the beginning of the process.
- RMSProp: modifies AdaGrad by changing gradient accumulation into an exponentially weighted moving average. This has the effect of discarding history from distant past so that it can converge rapidly after finding a locally convex area.
- Adam: combines RMSProp with momentum.

Cost Functions

- The choice of cost function is closely linked to the choice of output unit (choice of output determines the likelihood in the ML framework).
- Most modern NNs are trained using Squared Loss (and variations) or Maximum Likelihood (the cost function is the negative log-likelihood).
- One recurring issue with NNs is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small.

functions that become very flat are not good, as the gradient becomes very small

Output Units

saturation refers to a function being flat when values of x are large, like for sigmoid functions

Linear Units

- One simple and commonly used output unit is the linear unit $\hat{\mathbf{y}} = \mathbf{W}'\mathbf{h} + \mathbf{b}$. Because linear units do not saturate they pose little difficulty for the gradient-based optimization algorithms.

Binary Units

- Many common classification tasks involve predicting a binary y. The ML approach is to define a Bernoulli distribution over y conditioned on the features. ML is almost always the preferred approach to training sigmoid output units.

Hidden Units

Rectified Linear Units (ReLUs)

- ReLUs use the activation function $g(z) = \max(0, z)$. This is the de facto default activation function recommended for use with most MLPs.
- Because of the piecewise linearity, ReLUs preserve many of the properties that make linear models generalize well.
- The derivatives of a ReLU are large whenever the unit is active and the second order derivative is zero almost everywhere.
- One drawback is that they cannot learn on examples for which their activation is zero.
- Various generalizations try to deal with that, the most common being Leaky ReLU

$$g(z) = \max(0, z) + \alpha \min(0, z)$$

where α is fixed to a very small value like 0.01. This allows the hidden unit to have a small gradient even when the unit is not active.

Hidden Units

Sigmoid and Hyperbolic Tangent Units

- Prior to ReLUs most NN used the Logistic Sigmoid or the Hyperbolic Tangent (tanh) activation functions.
- In modern applications use of sigmoid functions is discouraged due to potential saturation of the gradients (unless undone by appropriate choice of cost function).
- The hyperbolic tangent activation function typically performs better than the sigmoid (because it is more similar to the identity function near zero).
- Recurrent NNs, some auto-encoders and other probabilistic models have requirements that rule out the ReLus family in which case sigmoid or tanh activation functions become more appealing or necessary.

Architecture

Number of Hidden Units and Layers

- Choice of the number of hidden layers is a matter of experience and application. More hidden layers allow for different kinds of feature extraction but at the cost of model complexity. Recently a so called "deep-networks" have achieved tremendous success in a number of artificial intelligence applications.
- It is advisable to err on the side of having too many hidden units. With too few units the model might not be flexible enough to describe the data. With too many hidden units we are in danger of overfitting; however as we saw this can be dealt with via regularization.
- Thus a common strategy is to set the number of units in the range of 5 to 100 and shrink the irrelevant ones to zero using regularization (and cross validation for the tuning parameter λ).

Architecture

Number of Hidden Units and Layers

- The universal approximation theorems (loosely speaking) states that a feedforward network with a linear output layer and a any commonly used activation function can approximate any function provided that the network is given enough hidden units.
- The fact that an MLP can represent the true data generating function does not mean however that it will be able to learn it from the data.
- In reality deeper networks reduce the number of units required to represent the desired function.
- Choosing deep MLPs is also motivated statistically. We can interpret deep architectures as expressing a belief that the function we want to learn can be represented in a number of step where each step uses the previous step's output and where the function is a composition of several layers of simpler functions. Empirical evidence indeed suggests that using deep architectures indeed expresses a useful prior over the space of functions the model learns.

Learning and Generalization

Methods for Generalization

- As we have mentioned before, the purpose of any Machine Learning method is not to learn a perfect representation of the sample but to build an accurate model of the statistical process which generates the data and thus predict well out of sample.
- In other words, the key question in machine learning is how to create models that will perform well not only on training data but also on new, previously unseen data.
- Such models are said to be able to "generalize" (from sample to test data).

Learning and Generalization

What is Generalization?

- Generalization methods usually work by putting soft or hard constraints on on the model.
- Often these constraints are designed to encode (i) prior knowledge or (ii) preference for a simpler model in order to avoid overfitting.
- Alternatively generalization takes the form of ensembles, that is combining multiple models that explain the training data.
- At the heart of generalization lies the bias-variance trade-off and the idea of optimally balancing squared bias and variance in order to maximize out of sample predictive power.

Learning and Generalization

Early Stopping

- When a model has the capacity to overfit the data training for a sufficiently long number of iterations will cause the validation set error to start increasing, while the test error is still decreasing.
- Early stopping is a simple algorithm that terminates training when the validation los has not improved over the best recorded for a number of pre-specified of iterations (and then the best previously attained parameters are used).
- One way to think about this is to view the number of iterations as a hyper-parameter. A common approach here is to divide the data set into three subsets; one for training, one for validation and one for testing. The validation subset is used to monitor out of sample performance.
- Once the validation error starts increasing (even though the training error is probably still decreasing) it is a sign that the model training is ready to be stopped.
- Early stopping is both effective and simple which explains its popularity. In business problems it is less commonly used due to the scarcity of data. The catch is that it requires a validation data set.

Learning and Generalization

Reguralization: Parameter Norm Penalties

- Many generalization methods work by limiting the capacity of models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J resulting in

$$\bar{J}(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

  where α is a positive hyper-parameter typically learned from the data.
- For neural networks we usually choose a penalty that leaves the biases unregularized. We denote the weights affected by regularization by $\mathbf{w}$. Additionally with neural networks we might want to have multiple α parameters, one for each hidden layer.
- The two common norms used are L^2 and L^1. The L^2 parameter norm penalty drives the weights closer to the origin by adding

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{\theta}||_2^2$$

  making the objective function

$$\bar{J}(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) + \frac{\alpha}{2}\boldsymbol{w}'\boldsymbol{w}$$

Learning and Generalization

Reguralization: Parameter Norm Penalties

- Another option is to use L^1 regularization by adding

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1$$

making the objective function

$$\bar{J}(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y}) + \alpha \sum_i |w_i|$$

- Compared to L^2, L^1 regularization results in a solution that is more sparse (a feature selection mechanism).

Regularization is a method to avoid high variance and overfitting as well as to increase generalization. Without getting into details, regularization aims to keep coefficients close to zero. Intuitively, it follows that the function the model represents is simpler, less unsteady. So predictions are smoother and overfitting is less likely (graphic below

Learning and Generalization

Dataset Augmentation

- The best way to make a model generalize better is to train it on more data.
- Since in practice the available data is limited one approach is to create fake data and add it to the training set.
- This is easier said than done; harder for regression problems and especially hard for business problems.
- Injecting noise in the input to a neural network can also be thought as a form of data augmentation. Noise can be applied to either the input or less obviously to the hidden units (dataset augmentation as multiple levels of abstraction).

Learning and Generalization

Noise Robustness

- Another way that noise has been used in regularization is by adding it to the estimated weights (primarily in the case of recurrent nets).
- Intuitively, making the model impervious to weight noise pushes the optimization to find solutions that are insensitive to small variations in the weights, thus finding points that are not only minima but also surrounded by flat regions.

Multitask Learning

- Multitask learning is a way to improve generalization by pooling multiple examples into one model.
- The idea is that when a part of a model is shared across multiple tasks, that part of the model is more constrained toward good values.

Learning and Generalization

Parameter Tuning & Sharing

- There may be ways to express prior knowledge about suitable values of the model parameters from domain knowledge.
- A common way to do this is to impose a constraint that certain parameters should be close to one another. In this situation we can impose a constraint

$$\Omega\left(\mathbf{w}^A, \mathbf{w}^B\right) = ||\mathbf{w}^A - \mathbf{w}^B||_2^2$$

- Another approach is to constrain the parameters of a supervised model to be close to the parameters of another unsupervised model.
- At the limit, this leads to parameter sharing, where sets of parameters are forced to be equal. The most common example of parameter sharing occurs in convolutional neural networks.

Learning and Generalization

Bagging and Ensembles

overfit error

- Bagging is a general technique for reducing generalization error by combining several models. It is an example of a general strategy known as ensemble-ing.
- Intuitively, the reason why this works is that different models will not likely make the same errors on the test set.
- The expected squared error of the ensemble predictor is

$$\mathrm{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k}v + \frac{k-1}{k}c$$

where $\mathrm{E}\left(\epsilon_i^2\right) = v$ and $\mathrm{E}\left(\epsilon_i \epsilon_j\right) = c$.

Learning and Generalization

Bagging and Ensembles

- Different ensemble methods construct ensembles in different ways.
- Bagging involves generating k different bootstrapped datasets. It is less obvious how to do this with time series!
- Neural networks can reach such a variety of solution points that they can benefit from ensemble-ing even if trained on the same data set, Difference in random initialization, selection of mini-batches, hyper-parameters, etc can often cause neural networks to have partially independent errors.
- Overall, model averaging using ensembles is an extremely powerful method to achieve good generalization performance. In fact machine learning competitions are usually won by ensemble methods. The downside is of course the increased computational burden.

Learning and Generalization

Dropout

dropout leaves some layers of the NN out

- Dropout is a computationally inexpensive but powerful method to regularize a large family of neural networks.
- In a sense dropout is a way to train and evaluate a bagged ensemble of exponentially many neural networks, while keeping the computational burden under control.
- Dropout works by dropping out units of the neural network at each training iteration, at random. Formally, denote by $\boldsymbol{\mu}$ a mask vector which specifies which units to include and now $J(\boldsymbol{\mu}, \boldsymbol{\theta})$ the adjusted cost function. Dropout consists of training $\mathrm{E}_{\boldsymbol{\mu}} J(\boldsymbol{\mu}, \boldsymbol{\theta})$.
- To train we typically use mini-batch based optimization algorithm but each time we load an example into a mini-batch we randomly sample a different binary mask to apply to all the hidden and input units in the network. Typically, a hidden unit is included with probability 0.5 and an input unit with probability 0.8.

Learning and Generalization

Dropout

- Compared to bagging, the models evaluated in dropout are less independent. Instead they share parameters which makes it possible to represent an exponential number of models.
- In bagging each model is trained to convergence but in dropout models are not trained but for a single step. However it is the parameter sharing causes the models to arrive at good solutions.
- A large portion of the power of dropout comes from the fact that noise is applied to the hidden units. Destroying hidden features allows the estimation process to still make use of all the knowledge about the input distribution.

Reading and Acknowledgements

Reading

- These notes follow closely "Elements of Statistical Learning" (Springer, 2009) by J. Friedman, T. Hastie and R. Tibshirani (referred to as ESL in the notes).
- Some of the figures in this presentation are taken from "Elements of Statistical Learning" (Springer, 2009) with permission from the authors: J. Friedman, T. Hastie and R. Tibshirani
- These notes also follow closely Neural Networks for Pattern Recognition (NNPP) by C. Bishop.
- These notes also follow closely Deep Learning by I. Goodfellow, Y. Bengio and A. Courville.