deep feedforward networks

- Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value.
- It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.
- If you think of the final layer, which can be linear, it takes a nonlinear transformation of the input $\mathbf{x}$, and thru a linear transformation, it gets the answer.
- a linear model is not able to represent the XOR function → convince yourself.
- Often in neural nets, the non-linearity is introduced by the activation function (eg. sigmoid).
- The default recommended activation function in modern neural nets is the ReLU → rectified linear unit. $ReLU(y) = max(0, y)$, where $y = w^T x + b$. One reason we suspect is the gradient exists everywhere when activated so learning can take place with good feedback.
- The convergence point of gradient descent depends on the initial values of the parameters.
- The nonlinearity of neural nets causes most interesting loss functions to become nonconvex
- we want to initialize the weights to small random values so as to give each neuron a chance of being activated or not.
- Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data and the model distribution. An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model.
- One recurring theme throughout neural network design 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. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate.
- For real-valued output variables, if the model can control the density of the output distribution (for example, by learning the variance parameter of a Gaussian output distribution) then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity. Regularization techniques provide several different ways of modifying the learning problem so that the model cannot reap unlimited reward in this way.
- Mean squared error and mean absolute error can lead to saturating units with bad feedback gradients. That's another reason they aren't used, in addition to cross entropy being a MLE formulation.
- Sigmoid + log likelihood loss gets around the saturating function (and hence bad gradients) problem → easy to derive this.
- To get a probability distribution of outputs (example, for image classification problems), we can use the softmax function:
  $softmax(x_j) = \frac{e_j^x}{\sum_i e_i^x}$.
- The term "soft" derives from the fact that the softmax function is continuous and differentiable. The arg max function, with its result represented as a one-hot vector, is not continuous or differentiable.
- We often want to perform multimodal regression, that is, to predict real values from a conditional distribution p(y | x) that can have several different peaks in y space for the same value of x. In this case,

a Gaussian mixture is a natural representation for the output. Neural networks with Gaussian mixtures as their output are often called mixture density networks.

- In practice one can safely disregard the non-differentiability of the hidden unit activation functions like ReLU, by just picking a left or right derivative at the point of non-differentiability.
- One drawback to rectified linear units is that they cannot learn via gradientbased methods on examples for which their activation is zero. Various generalizations of rectified linear units guarantee that they receive gradient everywhere by using something like $ReLU(z_i) = max(0, z_i) + \alpha_i(min(0, z_i))$, with a small value of α_i.
  - leaky ReLU → small value of α_i like 0.01
  - absolute value rectification → $\alpha_i = -1$
- Maxout units (Goodfellow et al., 2013a) generalize rectified linear units further. Instead of applying an element-wise function g(z), maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of these groups. This can be viewed as learning the activation function, with large enough number of pieces, maxout units can represent any convex function with arbitrary fidelity.
- Because each unit is driven by multiple filters, maxout units have some redundancy that helps them resist a phenomenon called catastrophic forgetting, in which neural networks forget how to perform tasks that they were trained on in the past.
- another function to use as activation could be tanh which is closely related to sigmoid because tanh(z) = 2*sigmoid(2z) - 1
- Deeper networks are often able to use far fewer units per layer and far fewer parameters, as well as frequently generalizing to the test set, but they also tend to be harder to optimize. This is a tradeoff and the ideal balance must be found for each task.
- These intermediate outputs are not necessarily factors of variation but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.
- Some architectures add skip connections from layer i to layer $i + 1$. these make it easier for the gradient (the feedback) to flow from the output layers to layers nearer the input layers.
- backpropagation is the method of computing the gradient in a neural network. this is basically an application of chain rule for differentiation to the neural network architecture. the way this is done → refer to the notes for the neural nets and deep learning book by Nielsen.
- While doing backprop, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient will need to choose whether to store these subexpressions or to recompute them several times. This is a time-space tradeoff.
- The amount of computation required for performing the back-propagation scales linearly with the number of edges in G, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition.
- symbolic backprop: add additional nodes to a computational graph describing how to compute derivatives. the returned graph has more nodes and computes the gradient when graph is run and only parts of the graph's gradient can be evaluated. the other way is to do symbol to number differentiation: input is a set of numerical values that are input to the network and the graph itself and output is a set of numerical values that is the gradient.
- Each operation is responsible for knowing how to backprop thru the edges in the graph that it participates in, so the backprop algorithm itself doesn't need to know any differentiation rules.
- computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations.

- Dynamic programming ideas help in backprop because during one run of backprop, we could be computing the expression for $\frac{\partial u_i}{\partial u_j}$ many times. if we compute this value once and cache it in a table, we will benefit.
- The memory cost can be $O(mn)$ where m is size of minibatch and n is number of hidden units, since we store the value of each activation for each training example in the minibatch.
- The field of automatic differentiation is concerned with how to compute derivates algorithmically.
- A lot of numerical computation tricks can be used to make backprop run with lesser numerical errors → one idea is to use algebraic simplification (like simple cancellation) so as to not compute terms that can cancel and thus avoid error introduced by those terms.
- Consider the matrix multiplication $ABCD$. if D is a column vector while A has many rows, the graph will have a single output and many inputs, and starting the multiplications from the end and going backward requires only matrix-vector products. This order corresponds to the backward mode. Instead, starting to multiply from the left would involve a series of matrix-matrix products, which makes the whole computation much more expensive. If A has fewer rows than D has columns, however, it is cheaper to run the multiplications left-to-right, corresponding to the forward mode. This is just to show that sometimes backward mode is better, sometimes not.
- DL makes second order methods hard to use because the Hessian explodes in size → $O(n^2)$ wth number of parameters in network → so is hard to both compute and represent in memory.
- Historical notes:
  - chain rule was invented in the 17th century!
  - critics including Marvin Minsky pointed out sever flaws of the linear model family such as its inability to learn the XOR function, which led to a backlash against the entire neural nets approach, which may have held back progress → don't always listen to what the big people say, follow intuitions, and try things quickly.
  - the book Parallel Distributed Processing introduced backprop.
  - modern deep learning renaissance began in 2006.
  - in 2006ish, unsupervised learning was used to support supervised learning. now its the opposite. we use supervised learning, which we've made a lot of progress on, to support RL and unsupervised learning.