(25) ■ **Boosting:** On tabular data, Gradient Boosting Machines are best/among the best (outperform Random Forest). Also note that with GBM you typically don't do cross validation because there is high computational expense. Most people use out of bag error or set aside a validation set; Instead of GridSearchCV consider RandomSearchCV. Overfitting in boosting is not difficult, and it is essential to perform hyperparameter tuning! Adaptive basis func (learning function)→ like logistic regression.
■ **Neural Networks:** (NON-LINEAR method) Fit adaptive basis function model where $f(x) = \beta_0 + \sum_{m=1}^{M} \beta_m \phi_m(x, \varphi_m)$ where $\phi_m(x, \varphi_m)$ is learned from the data and are other ML models (Logistic Regression models; compositional logistic models); feature learning → each $\phi_m(x, \varphi_m)$ are new engineered features. FFNN → weights go left to right to make predictions. Output layer is usually Linear Classifier/Regression (want simple). Hidden layers = output of a logistic regression.
■ **Notation:** Input: x_j , $j = 1, \ldots, p$. Output: y_{ik} , $i = 1, \ldots, n$, $k = 1, \ldots, K$. y_{ik} → indicators for multiclass classification. Hidden units (neurons) z_{im} , $m = 1, \ldots, M$.
■ **Theory:** $f_k(x) = g_k(\beta_{0k} + \sum_{m=1}^{M} \beta_{mk} z_m)$ and $z_m = h(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{mj} x_j)$. Parameters: α_{0m} , β_{0k} = intercepts sizes M, K ; α_{mj} are weights going from input feature j to neuron m (size Mp) ; β_{mk} are weights going from the m^{th} neuron to the k^{th} output (size MK). Activation functions = g_k and h . g_k is the output activation. In classification, we want g_k to output prob of class k, use softmax $g_k(a) = \frac{e^{ak}}{\sum_{k=1}^{K} e^{al}}$ such that g_k: $\mathbb{R}^M \to [0,1]$. $h()$ is the activation of the hidden units (which are nonlinear). NOTE: the iterations is called the epoch.

■ **h() Activations:** ■ **Sigmoid function**: (issue w/ GD b/c flat parts – get stuck) $h(a) = \sigma(a) = \frac{1}{1+e^{-a}}$ (smooth and differentiable approx to spiking "neuron" off 0 or on 1. If $h(a) = \sigma(a) + g_k()$ is SoftMax then our NN is equivalent to Compositional Logistic Regression Model. ■ **ReLU** (Rectified Linear Units): $h(a) = \max(0, a)$, many neurons get shut off if their activation is less than 0. Leads to faster optimization b/c easy gradient and is less prone to vanishing gradient (getting stuck). Get sparse neuron leading to \ good interpretation, leaky ReLU doesn't get stuck but loses the sparsity. ■ **Leaky ReLU:** $h(a) = \max(0, a) + \alpha \min(a, 0)$, neuron is never shut off entirely.Regression = identity; classification = softmax
■ **Intro to Fitting NN:** How: MLE (called Empirical Risk Minimization in NN) and Loss (called Risk fct) $R(\theta)$; $\theta = [\alpha_{0,} \beta_0, \alpha, \beta]$. In Regression use MSE $R(\theta) = \sum_{i=1}^{n} \|y_i - f_k(x)\|_2^2$. In Classification use Cross-Entropy $R(\theta) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log f_k(x_i; \theta)$. NOTE: intercept = "biases" and parameters = "weights".
■ **Universal Approximation Theorem:** (Deeper NN = more expressivity!) Any MLP w/ enough hidden layers + units + non-linear activation can approximate any non-linear, bounded + continuous function $f(x)$ to an arbitrarily small error with high probability!

■ **Optimization:** $\min_{\theta} R(\theta)$. Not convex, we have parameters multiplying each other, there are many local minima, as the parameters increase the # of local minima increases. Algorithm is Gradient Descent → $\theta^t = \theta^{t-1} - \eta_t \nabla f(\theta^{t-1})$ GD update where η_t is the learning rate (step size). Since $f(x) = g(h(x))$ just use chain rule! Lead to fast computation through back propagation.■ **Why NN?** Expressivity. Very flexible estimates of $f(x)$. # of hidden units can be thought of a dim reduction, more hidden units the more expressive.
(26) ■ **NN:** (1) Want activation to be nonlinear so NN is nonlinear. (2) ReLU solves vanishing gradient problem that sigmoid has but prone to local solutions – leaky ReLU solves this local solution issue (gives slightly better solutions). (3) Going deep → multiple hidden layers = L (hidden = $L - 1$). Multiple hidden layers + fully connected/dense FFNN = Multi-Layer Perceptron (MLP) → gives huge number of parameters!
■ **Deep NN Properties:** (0) NON-CONVEX (1) Can fit higher order interactions. (2) # parameters explode = computationally expensive. (3) need tons of training data.(4) Architecture Hyperparameters = # hidden layers ; # hidden units per layer. (very data dependent and require tuning – don't want too small of either b/c it can lead to underfitting).
■ **Overfitting:** Benign overfitting – Even after training error = 0, we can still drive the test error down. → Implicit Regularization: Even when we don't explicitly penalize model complexity, the way the algorithm learns biases it toward simpler or more generalizable solutions. ■ **Fitting Strategy:** Gradient Descent. Approach: Backpropagation.
■ **Example:** $R(\theta) = \|y - f\|_2^2$; $f_k(x_i) = g_k(\beta_k^T z_i)$; $z_i = h(\alpha_m^T x_i)$. Gradient Updates: $\beta^{(t+1)} = \beta^{(t)} - \eta_t \frac{\partial R_i}{\partial \beta}$; : $\alpha^{(t+1)} = \alpha^{(t)} - \eta_t \frac{\partial R_i}{\partial \alpha}$. Idea: Chain rule→ $\nabla f(x) = \nabla g\big(h(x)\big) * \nabla h(x)$. $\frac{\partial R_i}{\partial \beta_{mk}} = -\big(y_{ik} - f_k(x_i)\big) \nabla g_k(\beta_k^T z_i) * z_{mi}$ = (weighted residual) * (m-th neuron)= $-\delta_{ik}^i * z_{mi}$. $\frac{\partial R_i}{\partial \alpha_{mj}} = -\sum_{k=1}^{K} \big(y_{ik} - f_k(x_i)\big) \nabla g_k(\beta_k^T z_i)\, \beta_{mk} * \nabla h(\alpha_m^T x_i) x_{ij}$ = (wgh res) * (j neuron) .
■ **Message Passing:** The gradients on the l^{th} layer only depend on weights/neurons directly connected in architecture diagram.
■ **Backpropagation:** (1) Forward Pass: fix all weights + compute neurons, estimate $f_k(x_i)$ and the residual δ_k^i . (2) Backwards Pass: starts with last layer and computes the residuals and the updated gradients and weights backwards to first layer!
■ **WHY is this important/done?** This is dome because the gradient updates are separable in each observation i and each neuron/weights meaning they can be updated in parallel – can distribute computation (using GPUs)!

(27) ■ **Deep Learning Optimization: (1) Gradient Descent**: $\theta^{(t+1)} = \theta^{(t)} - \eta_t * \frac{1}{n} \sum_{i=1}^{n} \nabla R_i(\theta^{(t)})$. When $n < 1000$ don't fit MLP, but for $n > 1000$ running Gradient Descent is expensive! So, need another solution **(2) Stochastic Gradient Descent (SGD)**: compute gradient not of average of all observations but a SINGLE observation! $\theta^{(t+1)} = \theta^{(t)} - \eta_t * \nabla R_i(\theta^{(t)})$.
■ **Results:** SGD does converge but very slow compared to gradient descent! Converges with a decreasing step size – need more iterations! INTUITION: converges because it is unbiased but with high variance, so, we need many iterations to converge. ■ **WHY SGD > GD:** SGD with GPU done in parallel can be much faster than GD (GD cannot be done in parallel).
■ **Popular Optimizations other than SGD:** (1) **Minibatch** = Averaging subset → gradient of subset of observations (stabilizes variance and remains unbiased). (2) Adaptive Learning Rate: Use different learning rates for different params θ – 2^{nd} moments of the gradient, e.g., **AdaGrad -** downside = learning rate can get very small which gives a very slow convergence. (3) **RMS Prop**: Use a moving average of the second moments. (4) **Momentum/Acceleration:** Working with nonconvex optimization problem so the idea is to avoid getting stuck in bad local minima → Polak's Heavy Ball Method. (5) **ADAM:** Uses minibatches, momentum and adaptive learning rates – benefits from lots of good properties - very good optimizer!!
■ **Results:** Different optimizers arrive at different local minima – since non-convex! Best ADAM (don't even need significant tuning) and RMS Prop (not so much SGD as it is slow to converge!). Choosing optimizer is like hyperparameter/training decision that must be made but in general just stick with ADAM.
■ **Avoid Overfitting in Deep Learning:** Goal: Benign overfitting where the training error = 0 but the test error still decreases. **Strategies: (1) explicit regularization** → ridge (called "weight decay"). **(2) Implicit Regularization** (all related to ridge and help avoid bad local optima)→ **(a) DROP-OUT:** "silence"/zero out a random subset of neurons at each iteration (kind of like random forest randomly subset of some features to split on). **(b) Augmentation:** Add more data/samples with noise to the training process (can help balance out classes, etc)! **(c) Early Stopping:** Stop at prespecified point – stop training early!
■ **Training Decisions:** (1) Deep learning is very dependent on initialization of parameters. Want to use **random** initializations. (2) Normalization (center + scale) → widely used in deep learning training, normalize the X's and y's before training (and normalize weights after updates).■ **Validation in DL:** Since there are so many hyperparameters and training takes so long, a validation set is essential (CV is too slow). Also report training and validation loss as a function of the Epochs to viz! Gridsearch is too expensive – wanted (1) **Guided Search** (use intuition to fix groups of hyperparams to reasonable defaults and fit over others in a greedy manner to find "good" set of hyperparameters. (2) Random Search: test random combs.