INFO 251: Applied Machine Learning

Neural Network Implementation

Key concepts (last class)

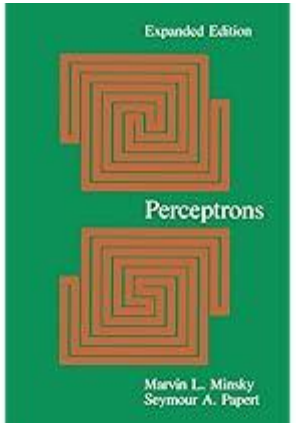
- Mimicking basic neural processes
- The perceptron
- Perceptron limitations
- Rosenblatt's algorithm
- Perceptron training vs. gradient descent
- Multilayer networks
- Universal approximation theorem

Outline

- **Learning multilayer weights: Intuition**
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- Practical considerations
- Summary

Multi-layer networks are great!

- Multi-layer networks have great properties, e.g. can solve the XOR problem – formally recognized by Minsky and Papert (1969)



- However, people didn't know how to fit these MLPs
  - In other words, how to adjust the weights and biases so that the network produces the correct outputs for given inputs?
  - The perceptron training rule we discussed doesn't work with multiple layers
  - Still, no one has figured out how to generalize perceptron training to multiple layers

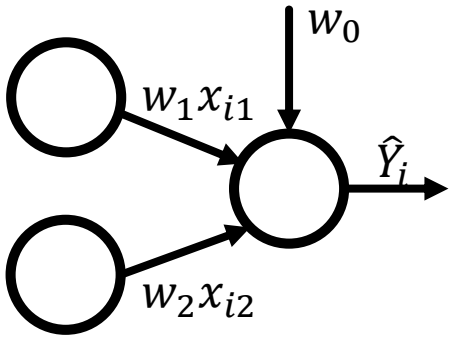
Intuition check 1

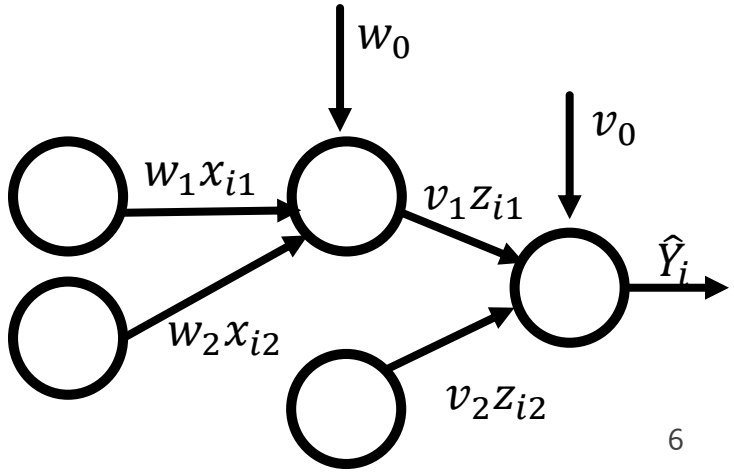
- Can off-the-shelf gradient descent, such as what you implemented in PS4, be used to learn the weights in a neural network?
  - No! The "activation" step function of a perceptron creates non-convex loss. Not differentiable

$$\hat{Y}_i = \begin{cases} 1 & \text{if } w_0 + w_1 x_{i1} + \cdots + w_n x_{in} > 0 \\ -1 & \text{otherwise} \end{cases}$$

Intuition check 2

- Well, if the issue is with the step function, can we just omit the step function entirely?
- No!
  - Without activation, each individual unit is linear
  - $\hat{Y}_i = w_0 + w_1 x_{i1} + \cdots + w_n x_{in}$
- Combination of units is more complex…
  - $\hat{Y}_i = v_0 + v_1(w_0 + w_1 x_{i1} + \cdots + w_m x_{im}) + \cdots + v_n x_{in}$
  - But still linear!



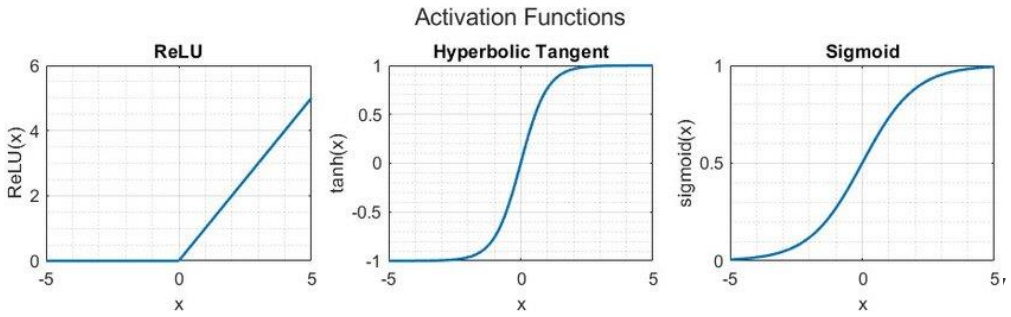


So, what do we do?

- We need something like a step function to capture non-linearities

$$\hat{Y}_i = \begin{cases} 1 & \text{if } w_0 + w_1 x_{i1} + \cdots + w_n x_{in} > 0 \\ -1 & \text{otherwise} \end{cases}$$

- But the step function itself creates issues for learning weights
  - It's nonlinear (good!), but not differential (bad!)
  - Gradient descent needs a differentiable function
- In other words, we need a nonlinear, differentiable function
  - E.g., sigmoid function, tanh(), ReLU

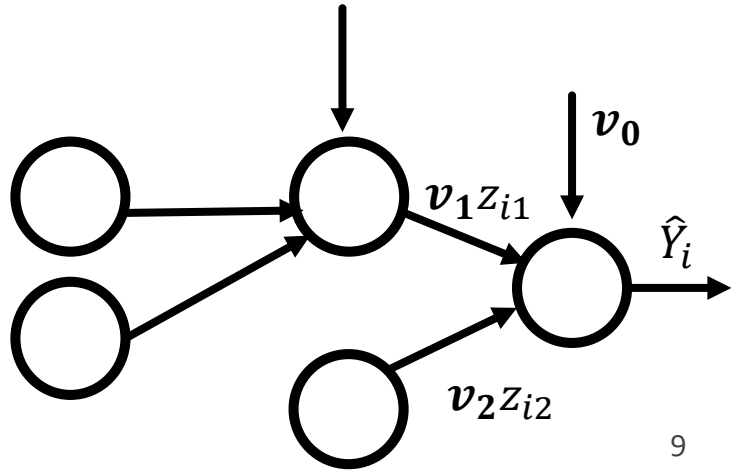


Outline

- Learning multilayer weights: Intuition
- **Generalizing logistic regression**
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- Practical considerations

Generalizing logistic regression

- Recall the loss functions from before
  - Logistic Regression (w/o regularization)
    - $J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} Y_i \cdot \log \hat{Y}_i + (1 - Y_i) \log\left(1 - \hat{Y}_i\right)$



  - In principle, something like this could work on the **last** (output) layer of a multi-layer network, to determine the weights on that layer
    - e.g., the bold $\boldsymbol{v_i}$'s in the diagram
  - What about the other layers?
    - Unfortunately, we don't have the target value for hidden layers!

Outline

- Learning multilayer weights: Intuition
- Generalizing logistic regression
- **Learning multilayer weights: simple case**
- Backpropagation: Intuition
- Backpropagation: Video
- Practical considerations

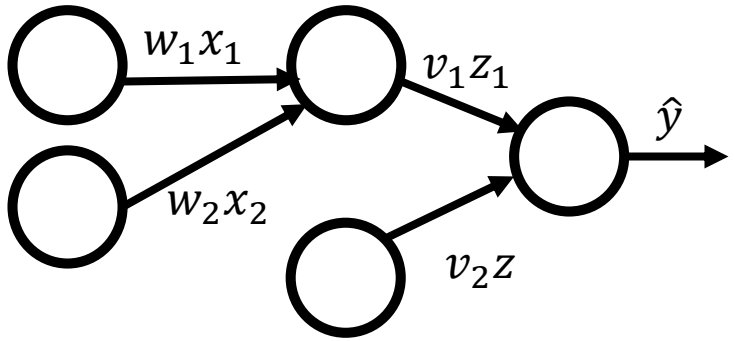
Two-layer backpropagation

- Backprop = gradient descent + chain rule
  - Solution and algorithm for two layers is not too complex (see Daume)
- Objective:

$$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\left(y_n - \underbrace{\sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)}_{\text{This is the prediction } \hat{y}_n}\right)^2$$

  - n indexes observations
  - i indexes hidden units
  - v is second layer weights
  - f is the sigmoid function
  - $\boldsymbol{w}_i$ is the vector of weights feeding into node i
  - $\mathbf{W}$ is first layer weights

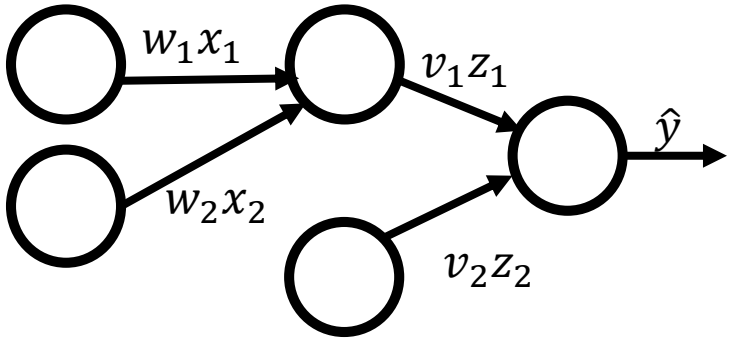
$$z_1 = f(w_1x_1 + w_2x_2)$$
$$z_2 = f(w_3x_1 + w_4x_2)$$



Two-layer backpropagation

- Objective:

$$\min_{\mathbf{W},v} \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2$$



- Loss:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right)^2$$

- Apply chain rule wrt. weights w:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial \boldsymbol{w}_i}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right) v_i = -ev_i$$

$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

prediction error $(y - \hat{y})$

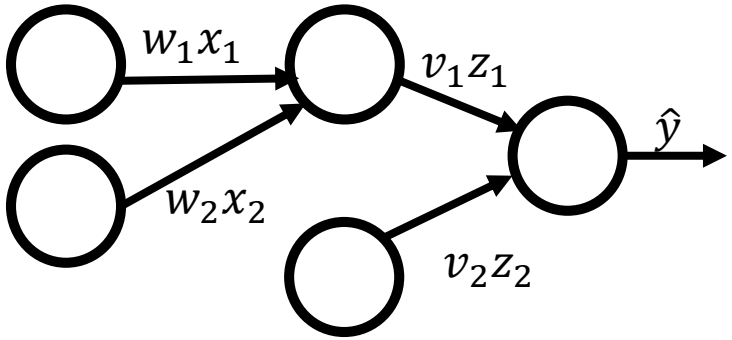
- In other words:

$$\nabla_{\boldsymbol{w}_i} = -ev_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$

Learning multilayer weights

- Solution with two layers

$$\nabla_{w_i} = -ev_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x}$$



- Does this make sense?
  - If predictive error (e) is small, take small steps
  - If v_i is small, hidden unit i has little influence on output, gradient should be small
  - If e or v_i changes sign, gradient should also flip sign

Outline

- Learning multilayer weights: Intuition
- Generalizing logistic regression
- Learning multilayer weights: simple case
- **Backpropagation: Intuition**
- Backpropagation: Video
- Practical considerations

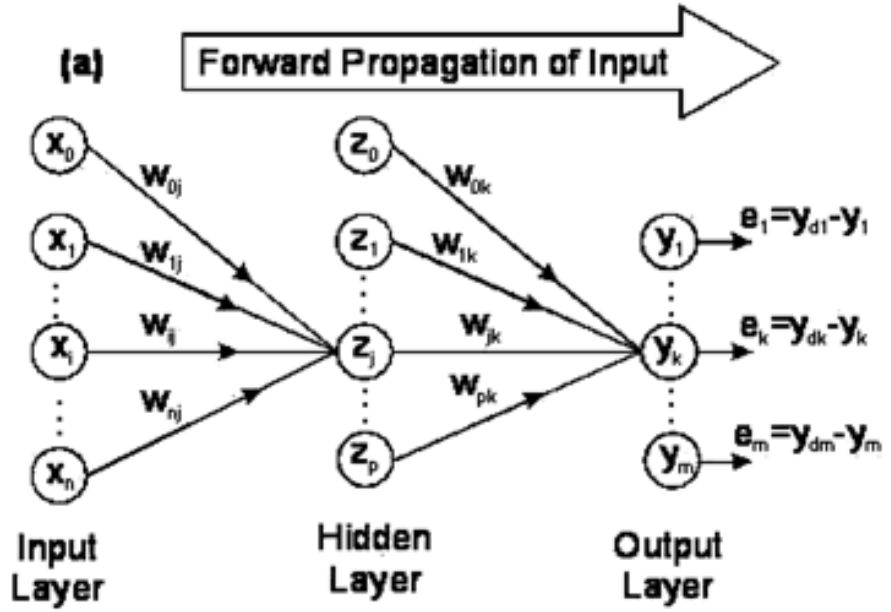
Backpropagation

- How to generalize from two layers?
- Sketch of procedure

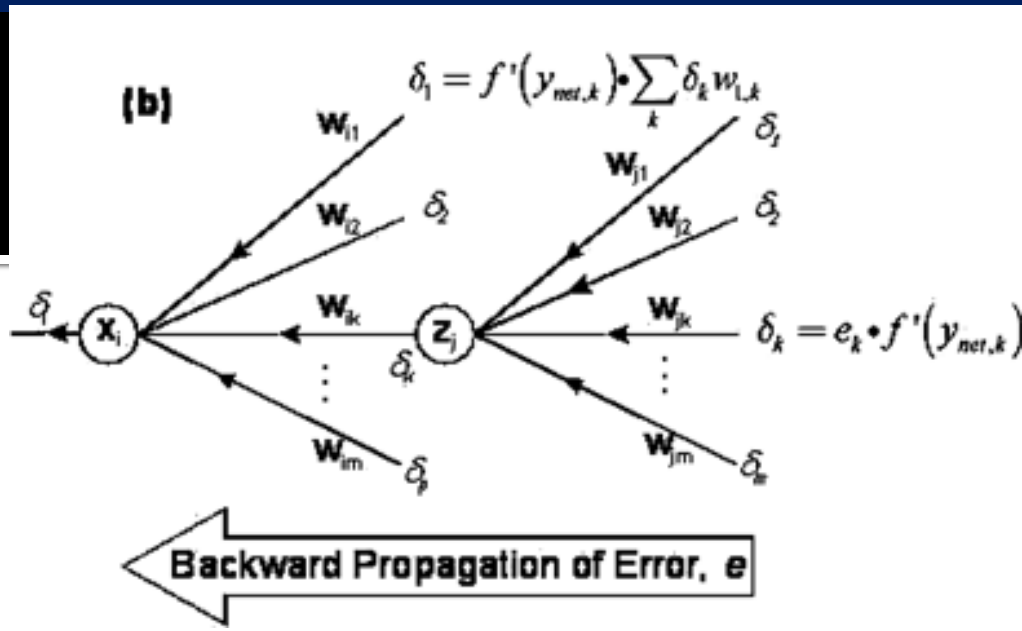
1. Forward Propagation -> Outputs
2. Backward Propagation -> Generate "deltas"
3. Weight Update -> same as in gradient descent

Intuition

- **Forward Propagation**
  - Given a training example (X_1 , …, X_n) and output Y_i:
  - Propagate inputs/activations forward, applying sigmoid function on dot products, calculate residuals



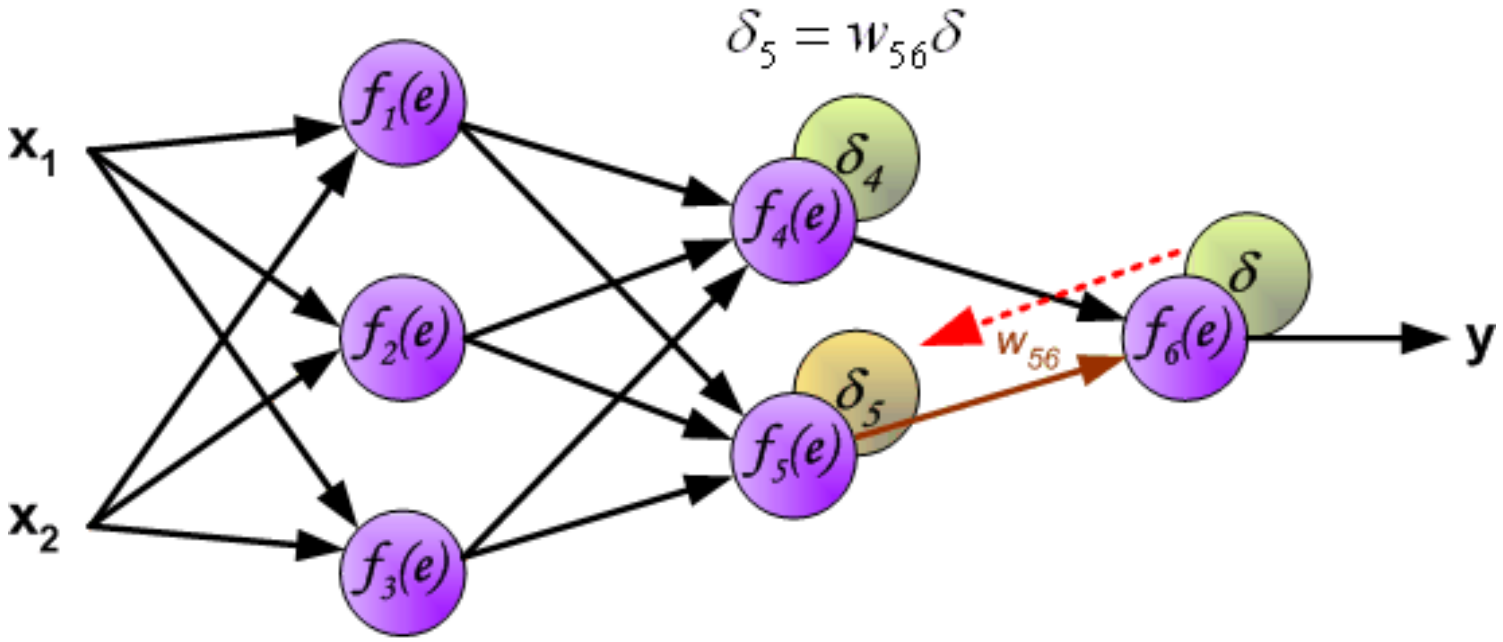
Intuition



- **Backward Propagation**
  - For a single training example i:
    - $\text{Cost}(i) = Y \cdot \log \hat{Y}_i + (1 - Y_i) \log(1 - \hat{Y}_i)$
    - i.e., how close is output to actual value?
  - Idea is to propagate costs backwards to earlier nodes
    - Compute δ_{jK} = "error" of j^{th} node in K^{th} (output) layer
      - $\delta_{jK} = Y_{jK}(1 - Y_{jK})(\hat{Y}_{jK} - Y_{jK})$
    - Note: deriving these partials is a bit complex, but mostly just chain rule + gradient descent (see ESL ch. 11)

Intuition

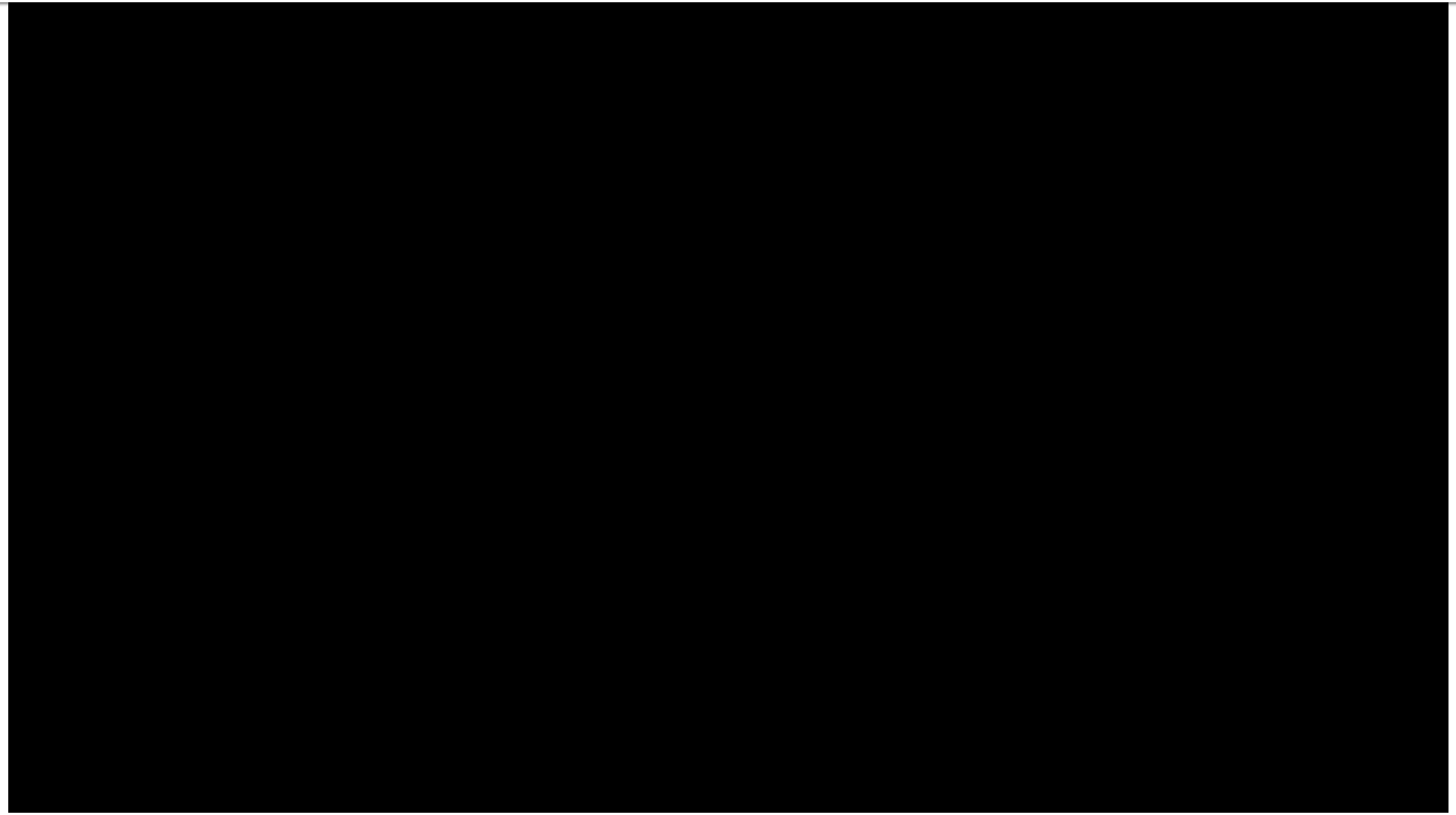
- **Update weights**
  - For each hidden unit *h* in k^{th} layer, update each weight as $+\eta\delta_{hk}x_i$



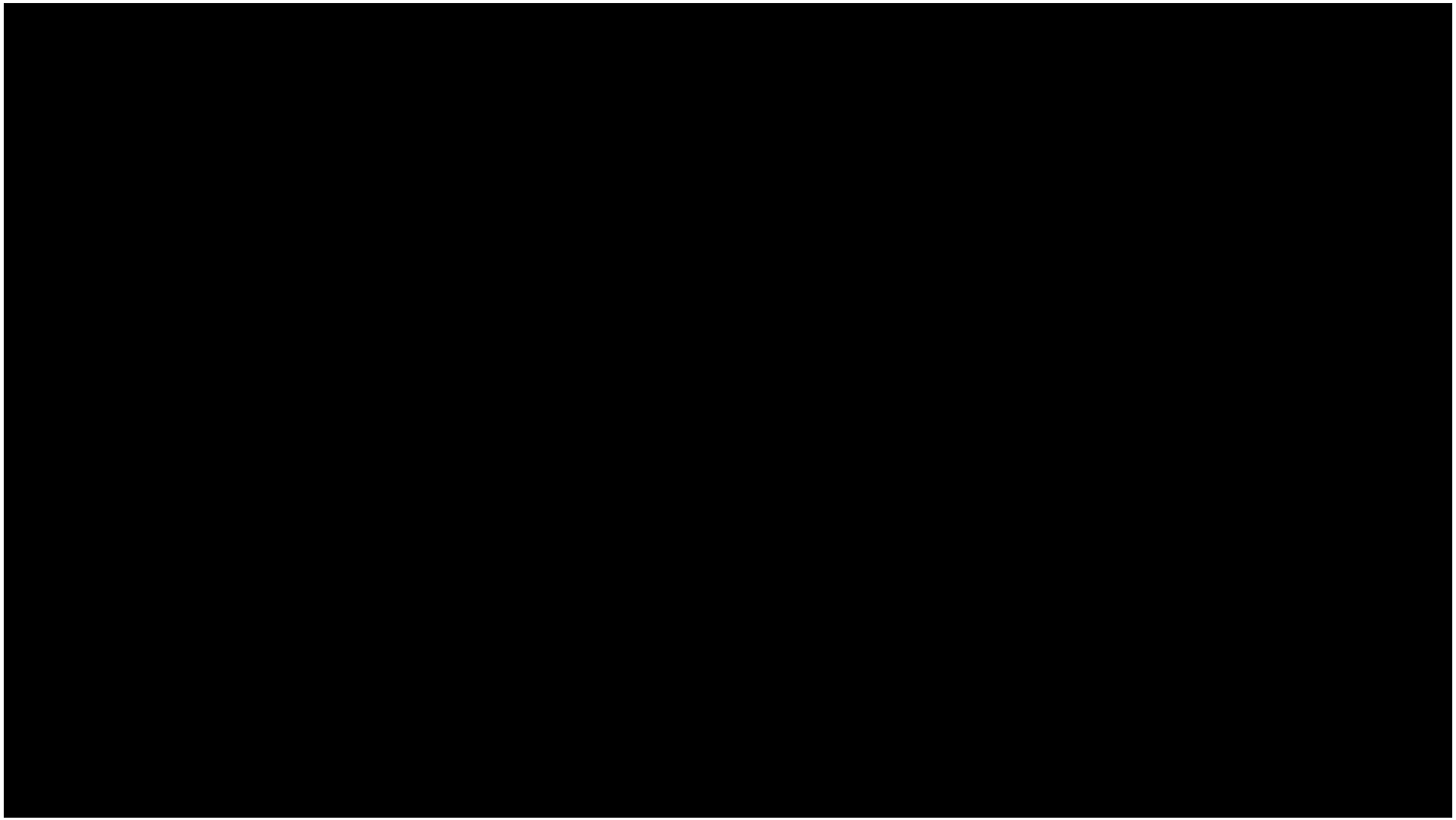
Outline

- Learning multilayer weights
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- **Backpropagation: Video**
- Practical considerations

What is back-propagation?



Video #2



Outline

- Learning multilayer weights
- Generalizing logistic regression
- Learning multilayer weights: simple case
- Backpropagation: Intuition
- Backpropagation: Video
- **Practical considerations**

Neural Networks: Issues

- Non-convex, sensitive to initialization
  - Common solutions: Randomize initialization (small random/uniform weights), train multiple networks
- Avoiding overfitting
  - Early stopping
  - Include fewer layers, weights per layer
  - Penalize large weights (weight decay with L1/L2 regularization)
  - Dropout – randomly drop neurons during training

Neural Networks: Issues

- Vanishing & Exploding Gradients
  - Problem: Gradients get too small (vanish) or too large (explode) in backprop
  - Solution: Activation functions like ReLU, Batch Normalization
- Choosing the Right Learning Rate
  - If learning rate too high, network doesn't converge; if too low, training is slow
  - Solution: Using learning rate schedules, adaptive optimizers (Adam, RMSprop)
- Computational Cost
  - Problem: Large networks require significant computational power
  - Solutions: GPU/TPU computing, model parallelization, gradient checkpointing, stochastic gradient descent (SGD), mini-batch training

Tuning networks: Not easy to get it right!

- Several hyperparameters to consider:
  - Learning rate – can tune, or use adaptive optimizers
  - How to initialize – randomize weights, train several networks
  - Regularization techniques – weight decay, dropout
  - Batch size – small batches can be noisy, large batches can be slow
  - How many layers – generalization vs. overfitting
  - How many units per layer – generalization vs. overfitting
  - When to stop?

Tuning networks: Semi-automated methods

- Grid search
  - Finds optimal hyperparemeters (within grid), but very computationally expensive
- Random search (of a subset of the grid)
  - Often does pretty well, and much faster, but can miss optimal configuration
- Bayesian Optimization
  - Uses probabilistic models to predict which hyperparameters will improve performance, focuses on tuning those – best when each model is costly to train
  - **Example**: If one learning rate produces poor performance, Bayesian optimization will avoid similar values and explore more promising areas
  - **Pro**: More efficient than grid and random search, converges faster to optimal values
  - **Cons**: Requires additional computation to model probability distributions

Tuning networks: Semi-automated methods

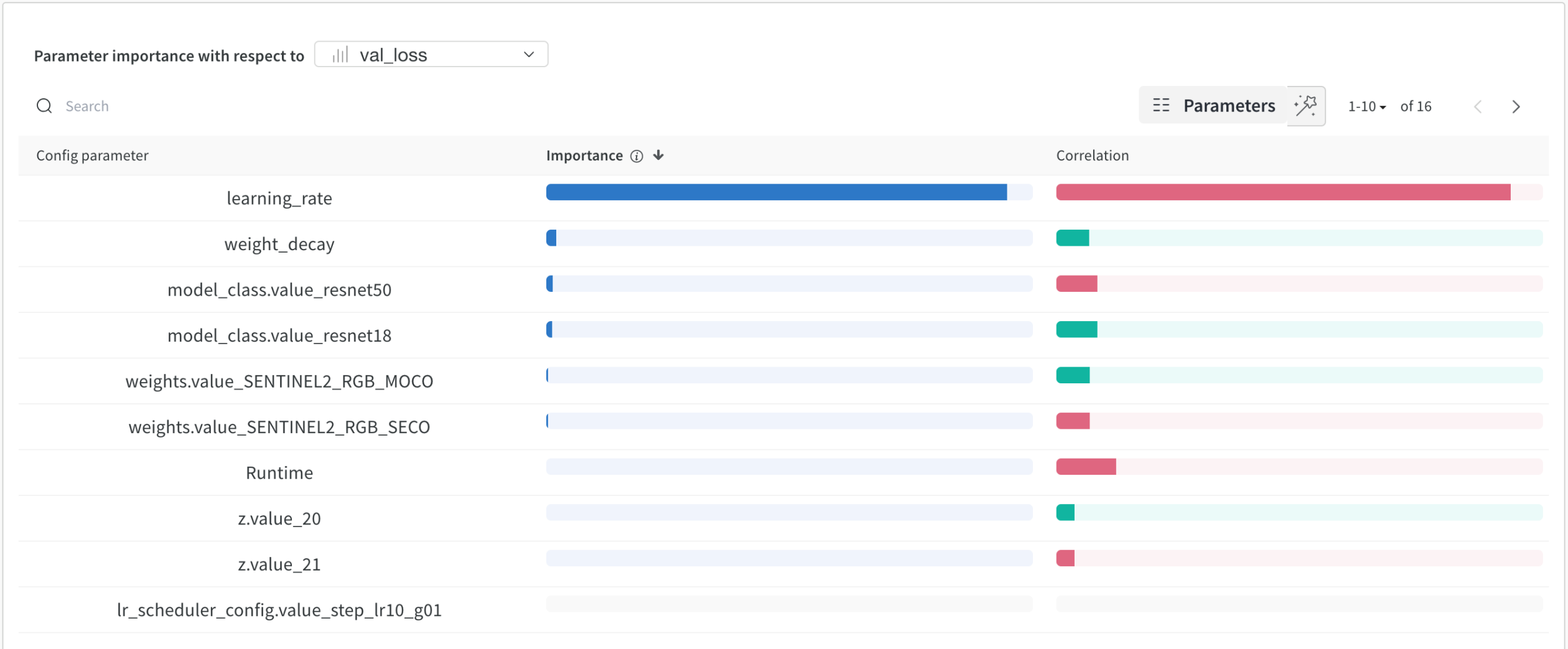
- There are other (less common) methods too!

Method	Best for	Pros	Cons
Grid Search	Small hyperparameter spaces	Exhaustive search, easy to implement	Computationally expensive
Random Search	Large hyperparameter spaces	Faster than grid search	Might miss optimal values
Bayesian Optimization	Efficient search for good results	Uses probabilistic modeling to find the best values efficiently	Computational overhead
Genetic Algorithms	Complex hyperparameter spaces	Can explore non-obvious configurations	Slow, needs many iterations
Hyperband	Large-scale deep learning tuning	Stops poor configurations early, saving time	Can discard good models too soon
Population-Based Training	Adaptive tuning for deep learning	Adjusts hyperparameters dynamically	Requires parallel computing
RL-Based Tuning	Complex, high-dimensional spaces	Learns best hyperparameters automatically	Requires expertise in reinforcement learning
Neural Architecture Search	Optimizing entire neural network designs	Finds optimal architectures	Requires massive computing power

Tuning Networks: Modern tools

- Bayesian optimization tools
  - Scikit-Optimize, Ax (Meta), Spearmint, GPyOpt, BayesianOptimization
- Hyperparameter optimization tools (more user-friendly)
  - Optuna, Hyperopt, Ray Tune, SKopt, SigOpt, Keras Tuner, GridSearchCV
- AutoML
  - Auto-sklearn, TPOT, FLAML, AutoKeras, Google AutoML, H20 AutoML
- Experiment monitoring and visualization
  - Weights and biases, TensorBoard, MLflow, Neptune.ai, Comet, Sacred

Tuning Networks: W&B



Satejsoman_berkeley's workspace | Personal workspace | Autosaved just now

Runs (10)

Search runs

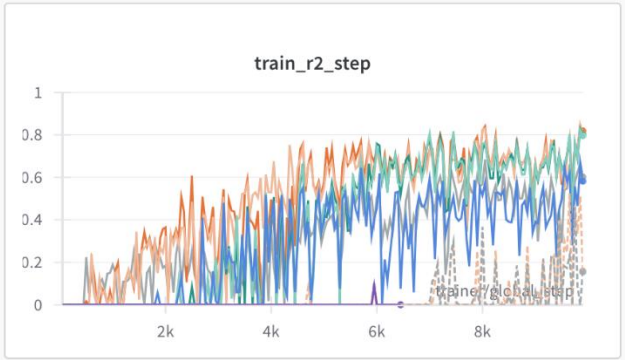
Search panels with regex | Settings | New report | Add panels

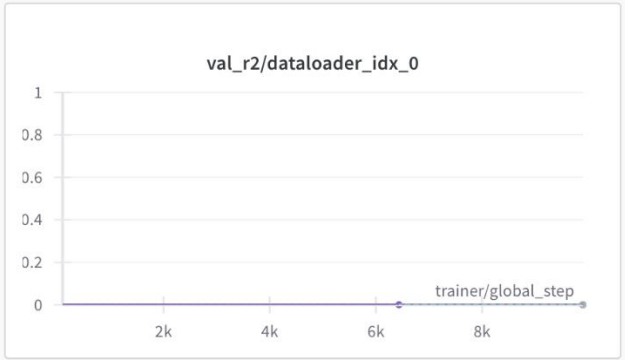
Name (10 visualized)

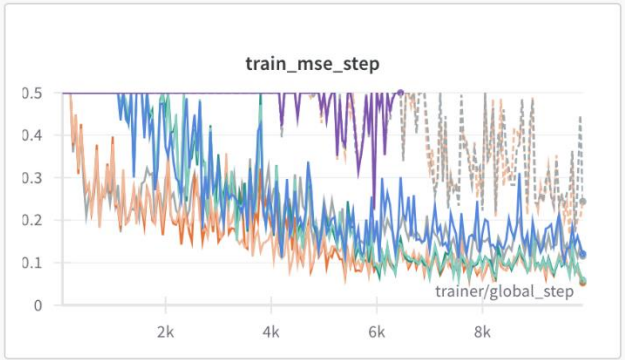
- lr: 0.005 3 4
  - wd: 0.1 1
  - wd: 0 2
  - wd: 0.01 1
- lr: 0.001 3 3
  - wd: 0.01 1
  - wd: 0 1
  - wd: 0.1 1
- lr: 0.0001 3 3
  - wd: 0.01 1
  - wd: 0.1 1
  - wd: 0 1

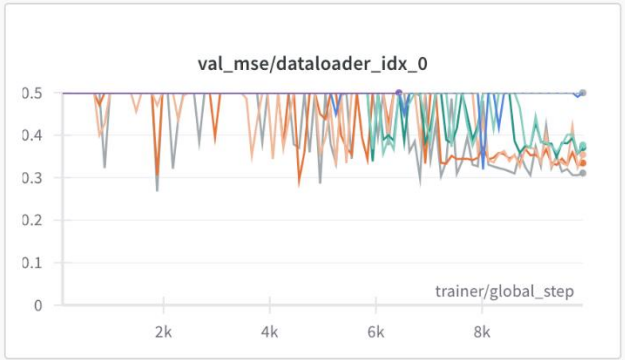
1-3 of 3

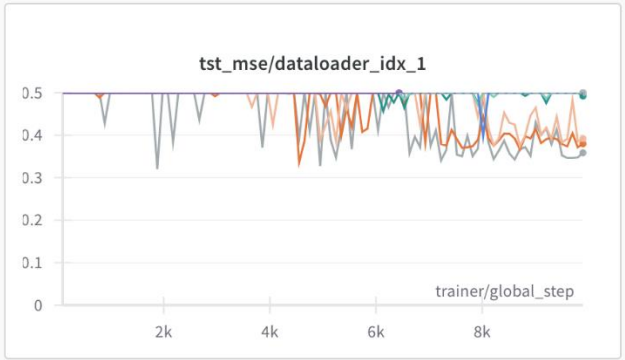
Charts 9 | 1-9 of 9

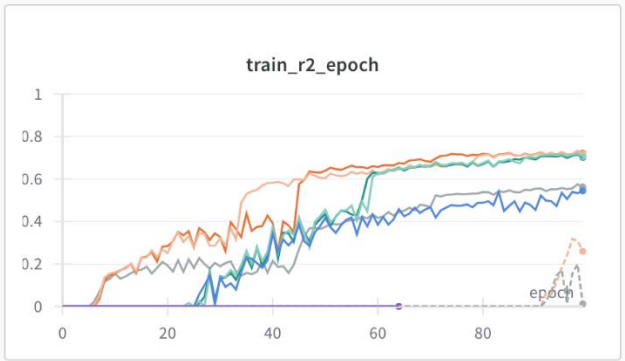


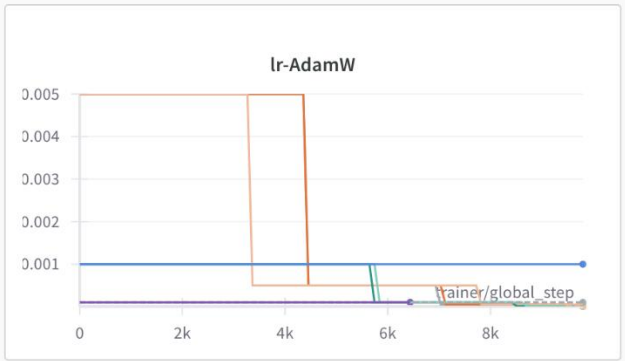


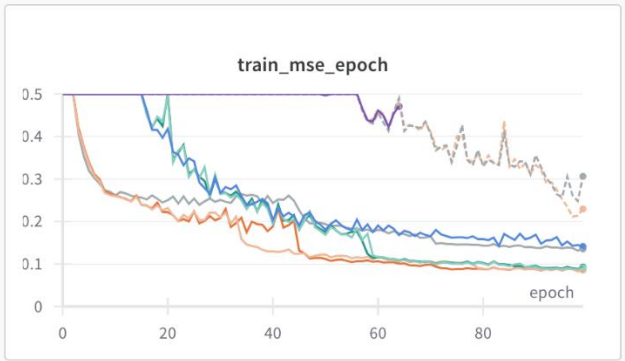












Tuning Networks: Best practices

- Start with "sensible" defaults

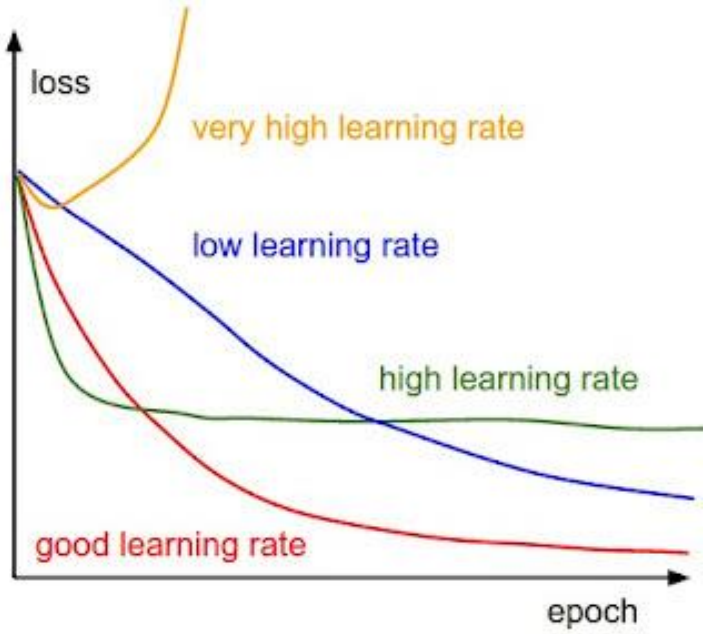
2.1 Common Default Values for Hyperparameters

Hyperparameter	Suggested Starting Values
Learning Rate	0.001 (Adam, RMSprop), 0.01 (SGD)
Batch Size	32 or 64 (small models), 128 or 256 (large models)
Number of Hidden Layers	1–3 (small tasks), 10+ (deep learning tasks)
Dropout Rate	0.2–0.5
Regularization (L2)	0.0001–0.01

- Then adjust based on results
  - Overfitting? (wedge betw/ train & eval)
    - Decrease complexity (reduce layers or neurons)
    - Increase regularization (L2 penalty or dropout)
    - Early stopping (halt training if validation loss stops improving)
  - Underfitting? (low performance on training data)
    - Increase model complexity (add more layers or neurons)
    - Reduce regularization (lower dropout or L2 penalty)
    - Train longer by increasing the number of epochs

Neural Networks: Summary

- Very flexible, can model complex and non-linear relationships
- Very hard to interpret, i.e. "black box"
- Tuning networks is not easy
  - Can use automated tuning methods
  - Helps to use reasonable defaults
  - Leverage learning curves and monitor loss to diagnose issues early



Key Concepts (this lecture)

- Multilayer networks
- Activation and non-convexity
- Why GD doesn't work on MLP's
- Backpropagaion
- Practical considerations
- Tuning networks