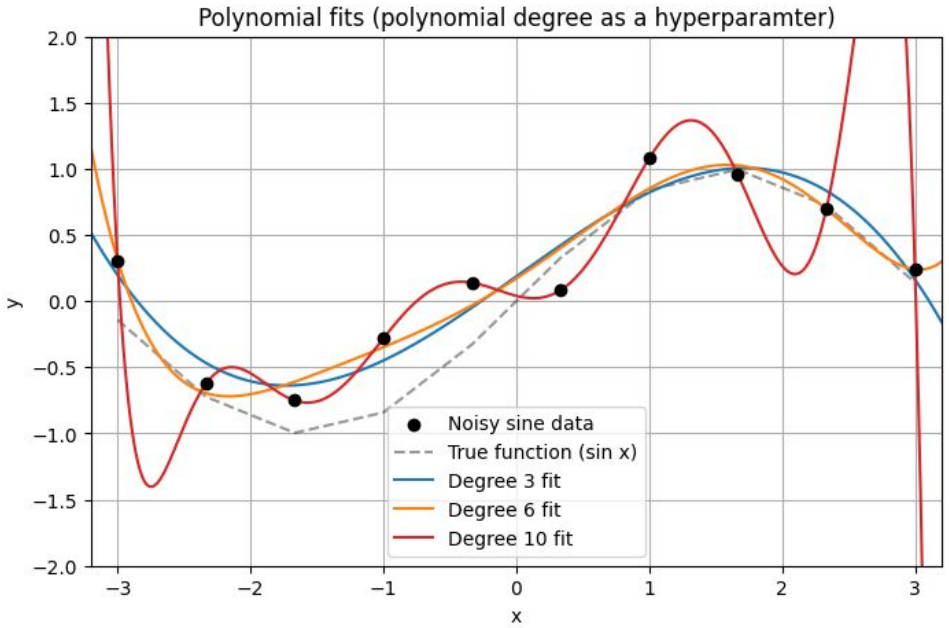
Hyperparameters and Optuna

Jack Carlton
University of Kentucky

What are Hyperparameters

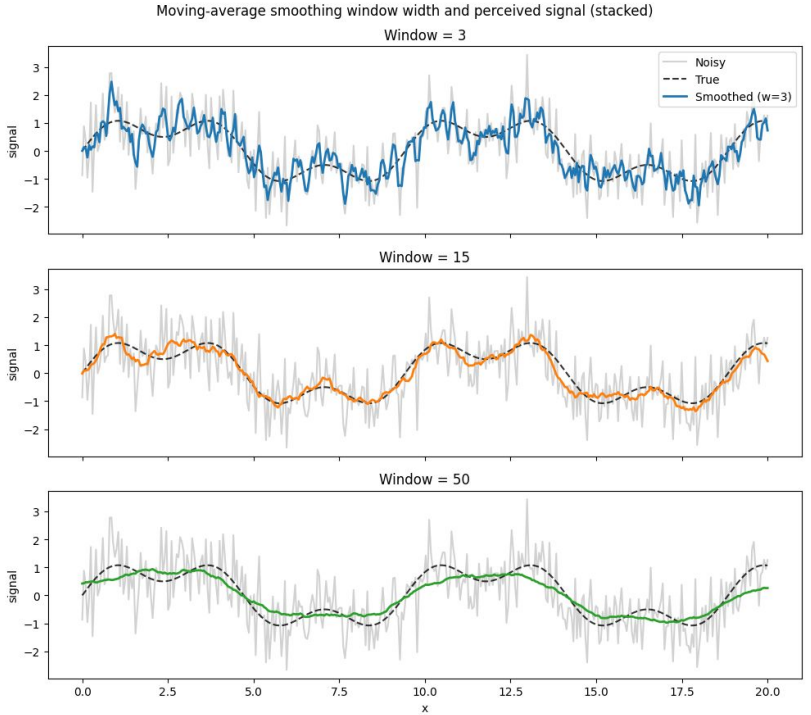
- **“A Hyperparameter is a parameter that can be set in order to define any configurable part of a model's learning process”**
  - These are parameters that are not learned by the model
  - Chosen before the model is trained
- Examples of what they control
  - Optimization dynamics (learning rate, batch size, optimizer)
  - Model capacity (hidden size, depth, heads)
  - Regularization (dropout, weight decay)



Example: Fitting data with a polynomial. The degree of the polynomial, d, is a hyperparameter

Hyperparameters Need to be Tuned Appropriately

- Improper hyperparameter choices can lead to
  - Overtraining
  - Slower convergence
  - Vanishing or exploding gradients
    - Impossible to find true solution at this point
- Tuning hyperparameters can be hard
  - They interact with each other (often non-linearly)
  - Optimal choice varies with each dataset/model

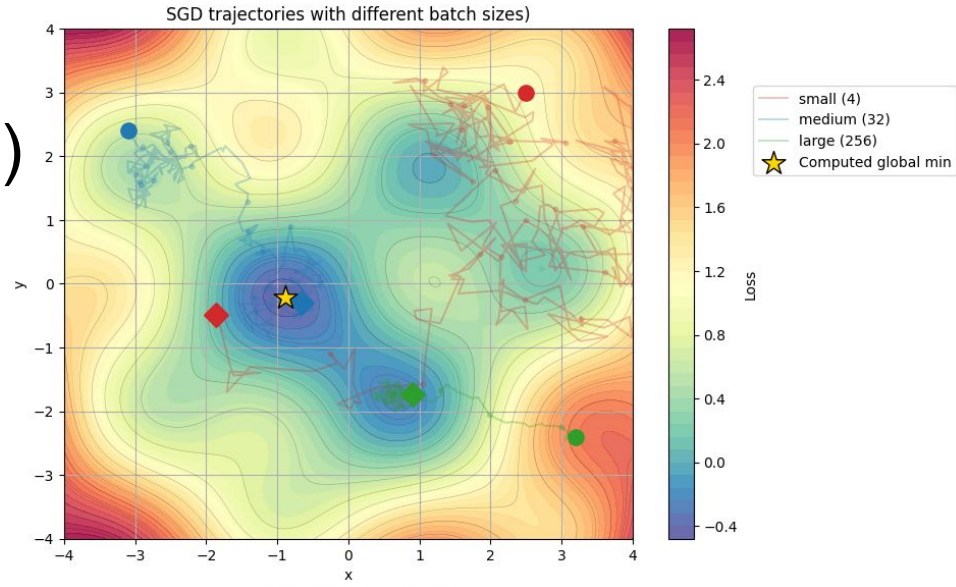


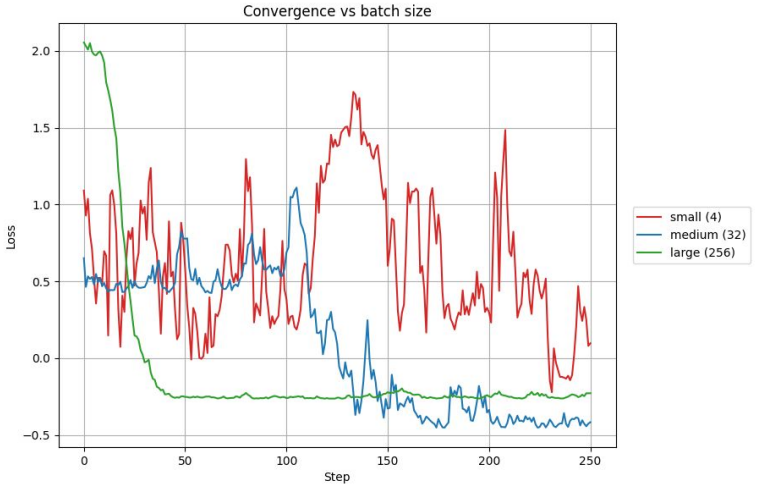
Example: Moving average window size is a hyperparameter. It can be over, or under tuned.

Hyperparameters for Neural Networks

Hyperparameters (Batch Size)

- More detail in this article
- **Batch size is how many training data points are processed before updating the model**
  - Makes predictions on batch, uses error to inform gradient descent algorithm
- Larger batch size
  - More accurate gradient descent
  - Fewer updates per epoch
  - Less noisy gradient
  - Less general, can get stuck in local minima
- Small batch size
  - Less accurate gradient descent
  - More updates per epoch
  - Noiser gradient
  - Can escape local minima





Hyperparameters (Learning Rate)

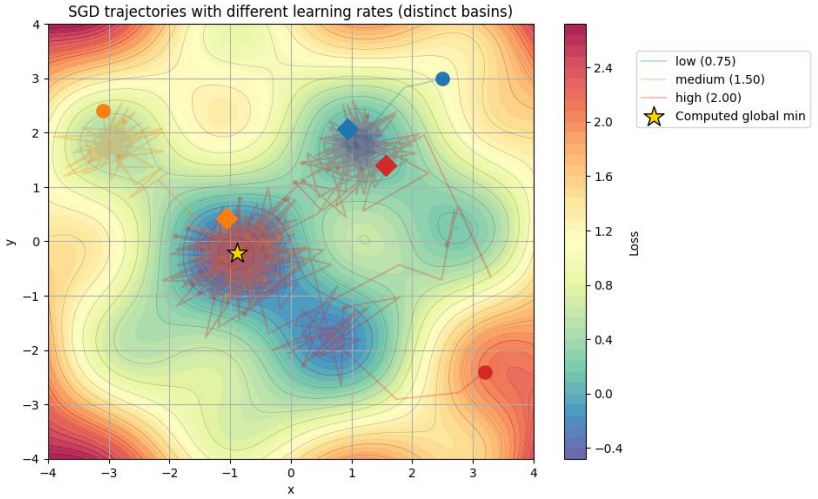
- More detail in this article
- **Learning Rate controls how large each parameter update is in the direction of the gradient**
- Larger learning rate:
  - Bigger, more exploratory steps
  - Less likely to get stuck in local minima
  - Harder to settle precisely at the optimum
- Smaller learning rate:
  - Smaller, more precise steps
  - More likely to get trapped in local minima
  - Better at fine-tuning near the optimum

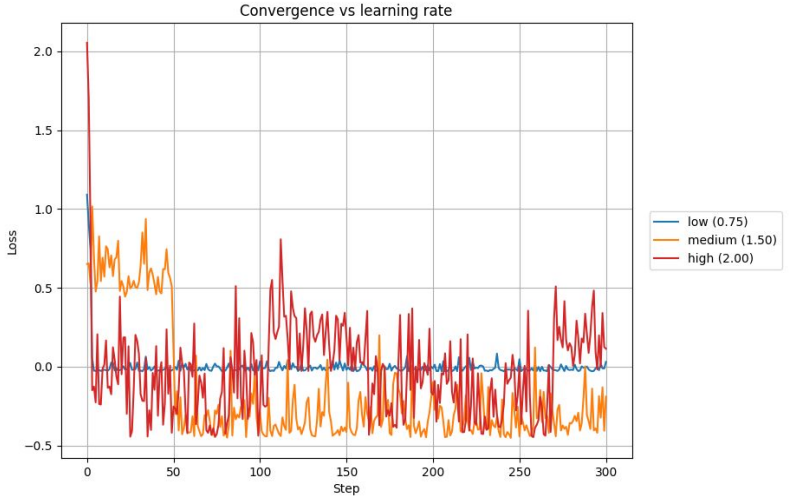
$$w_{\text{new}} = w_{\text{old}} - \alpha \nabla_w L(w)$$

w = model weights

α = learning rate

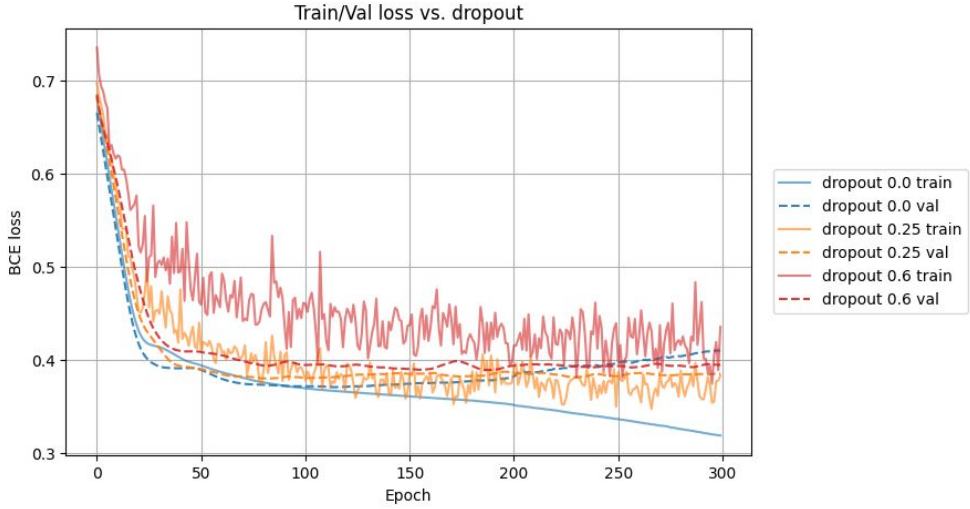
$\nabla_w L(w)$ = gradient of the loss with respect to the weights

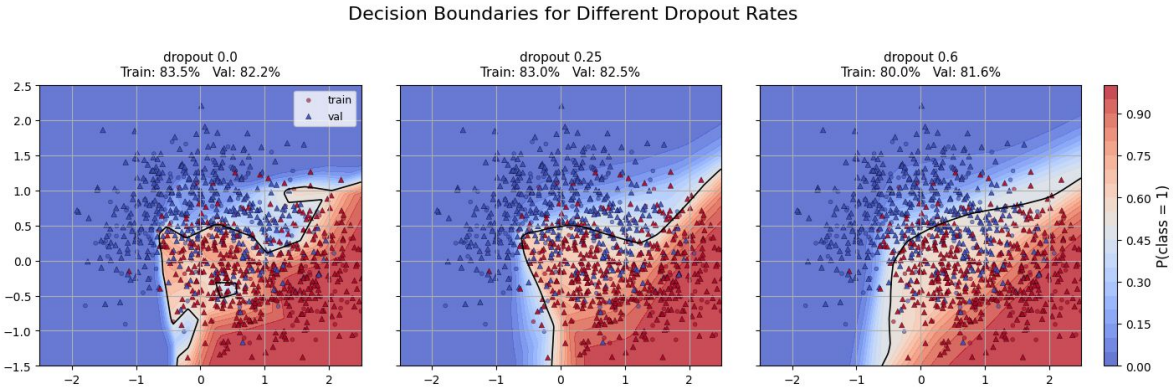




Hyperparameters (Dropout)

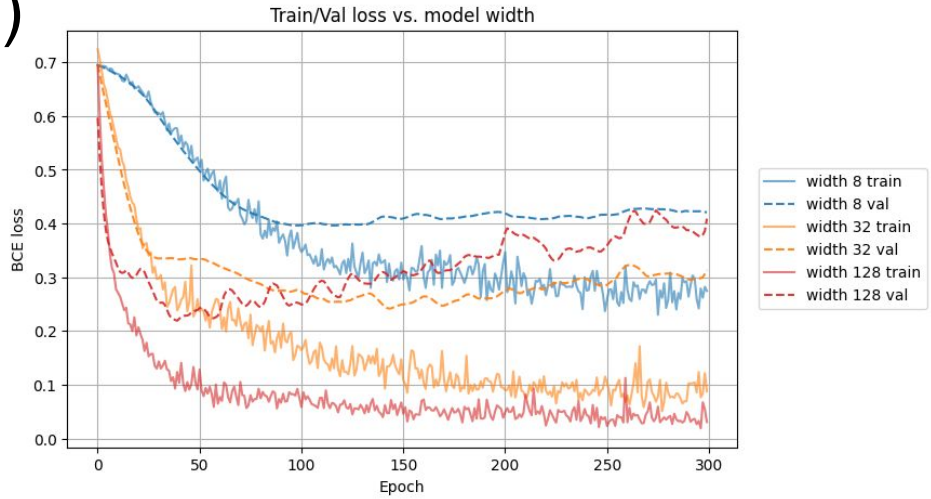
- More detail in this article
- **Dropout randomly disables a fraction of activations during training to reduce overfitting**
- Higher dropout:
  - Better for noisy datasets
  - Less risk of overfitting
  - Higher risk of underfitting
- Lower dropout:
  - Better for less noisy datasets
  - Less risk of underfitting
  - Higher risk of overfitting

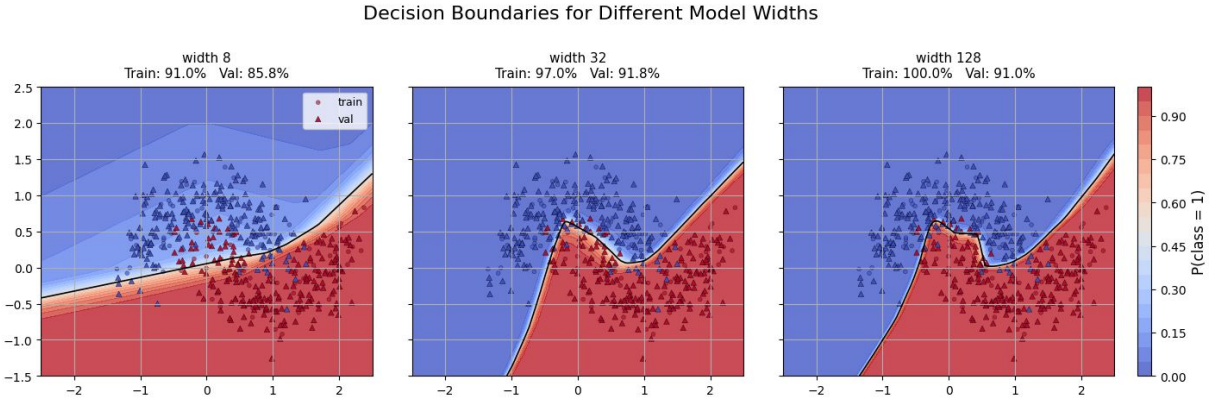




Hyperparameters (Hidden dim)

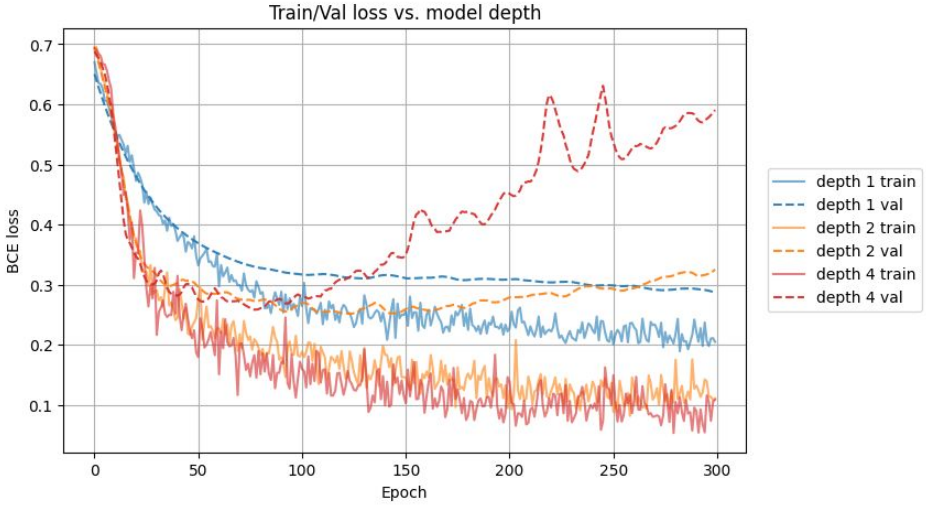
- More detail in this article
- **Hidden dimension sets the size of the model's internal feature representations at each layer.**
- Larger hidden dim:
  - Can represent more complex patterns
  - Higher risk of overfitting
  - Higher memory and compute cost
- Smaller hidden dim:
  - Limited ability to model complex patterns
  - Lower risk of overfitting
  - Lower memory/compute cost

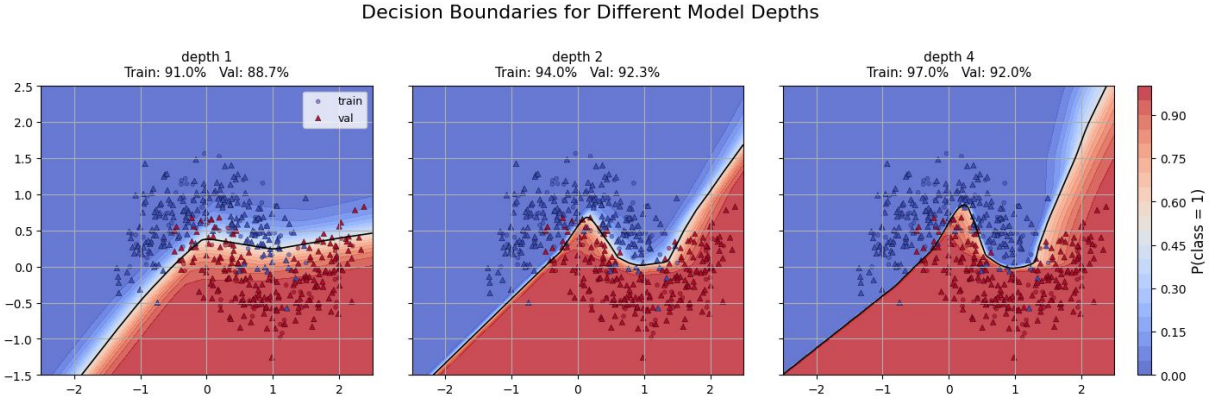




Hyperparameters (# of layers)

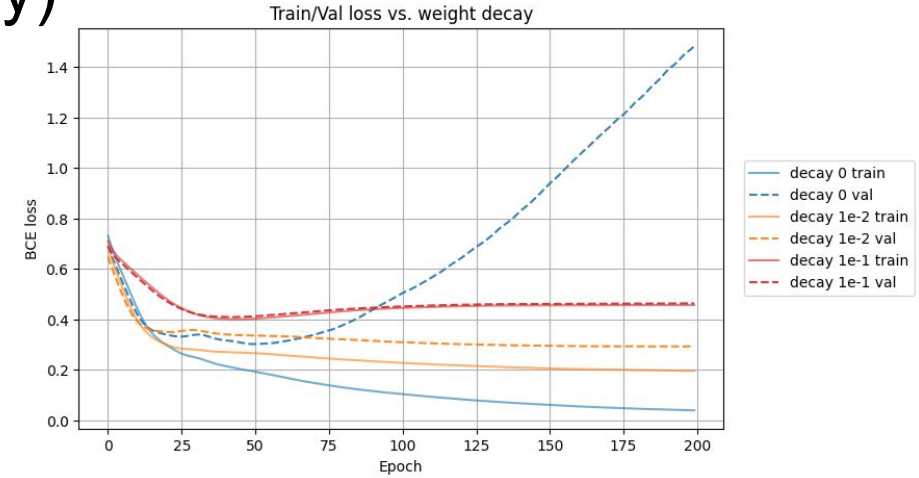
- More detail in this article
- **Hidden dimension sets the size of the model's internal feature representations at each layer.**
- More layers:
  - Learns more "hierarchical"/ "abstract" features
  - Higher risk of overfitting or training instability (higher loss fluctuations)
  - Higher memory and compute cost
- Less layers:
  - Learns "simpler"/"shallow" features
  - Lower risk of overfitting
  - Lower memory/compute cost

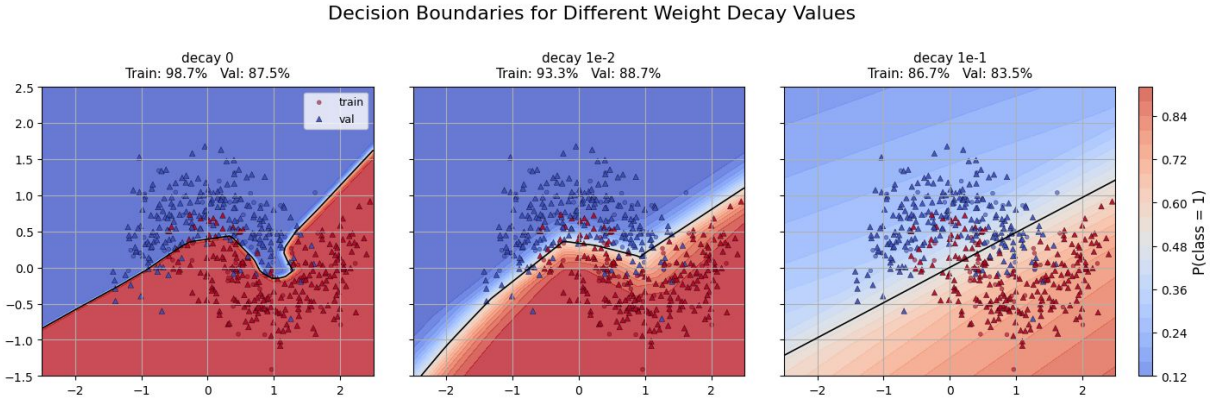




Hyperparameters (weight decay)

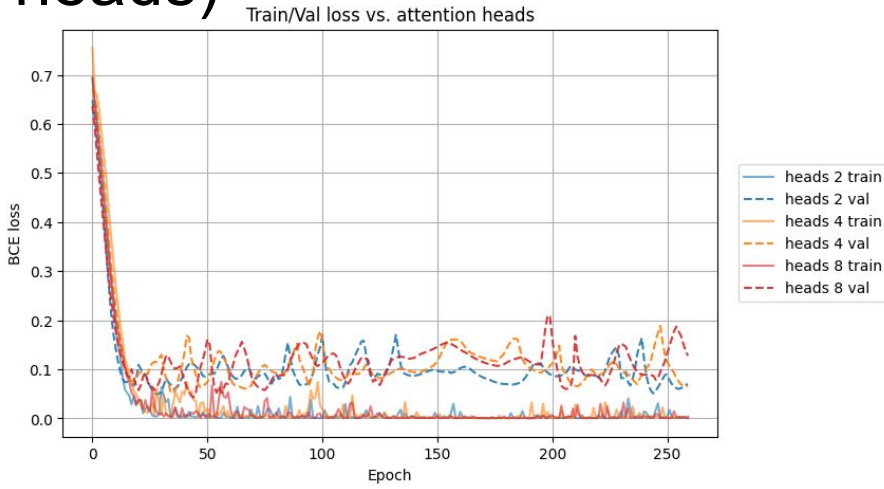
- More detail in this article
- **Weight decay adds a penalty on large weights to the loss, encouraging simpler models. Similar to dropout.**
- Higher weight Decay:
  - Better for noisy datasets, stronger regularization
  - Lower risk of overfitting
  - Higher risk of underfitting
- Lower weight decay:
  - Better for less noisy datasets, less regularization
  - Lower risk of underfitting
  - Higher risk of overfitting

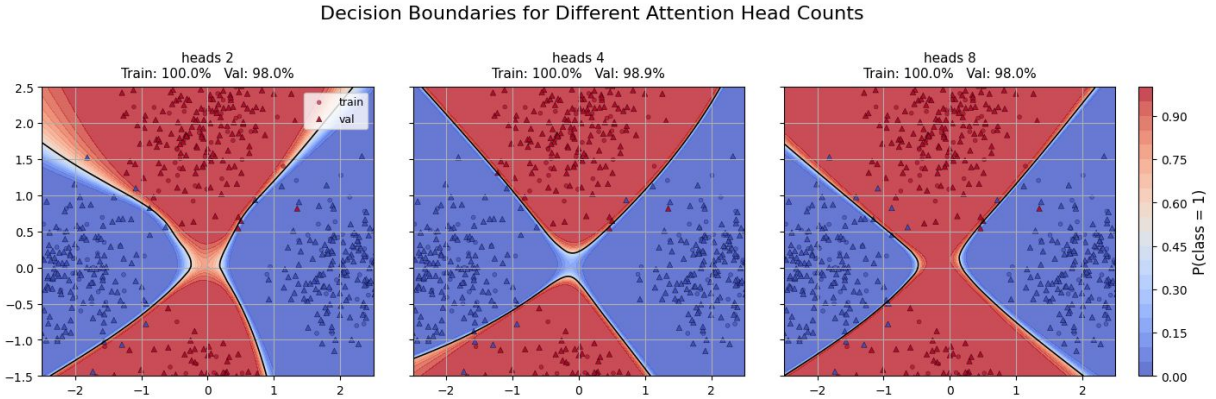




Hyperparameters (# of attention heads)

- More detail in this article
- **The number of attention heads controls how many independent subspaces the model attends to in parallel within each attention layer**
- More attention heads:
  - Learns multiple attention patterns in parallel
  - Higher memory/compute cost
  - Diminishing returns if per-head dimension is too small
  - Can overfit and destabilize training
- Fewer attention heads:
  - Learns fewer attention patterns
  - Lower memory/compute cost
  - May miss distinct relationships
  - Can underfit if data is very heterogeneous



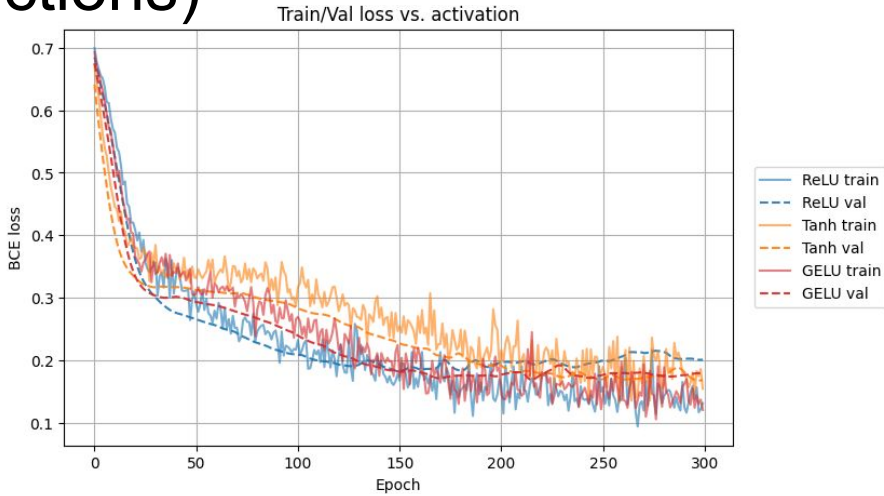


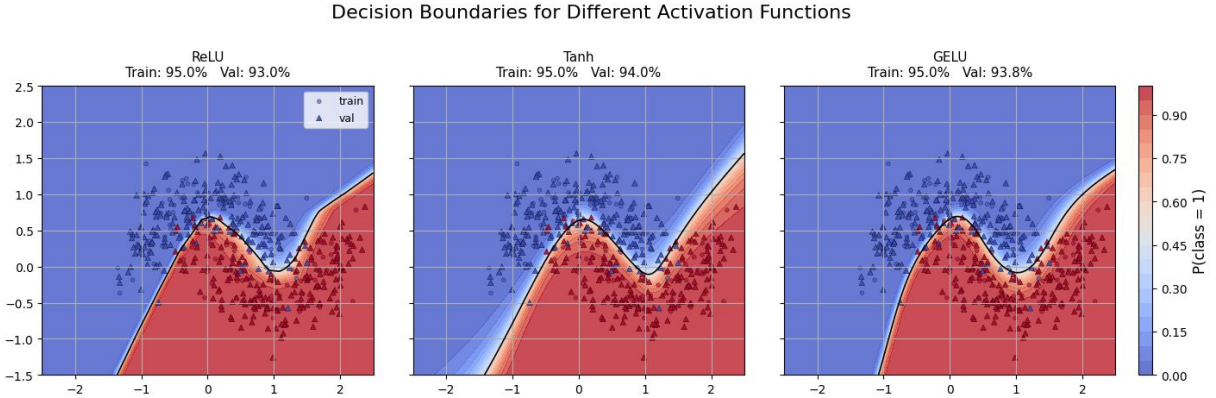
Hyperparameters (activation functions)

- More detail in this article
- **The activation function controls the nonlinearity applied at each layer**

Examples:

- ReLU
  - Simple, piecewise-linear activation
  - Fast training; can form sharp, noisy boundaries
- Tanh
  - Smooth, bounded activation
  - Produces smoother boundaries; may saturate
- GELU
  - Smooth, probabilistic gating
  - Balances smoothness and expressiveness



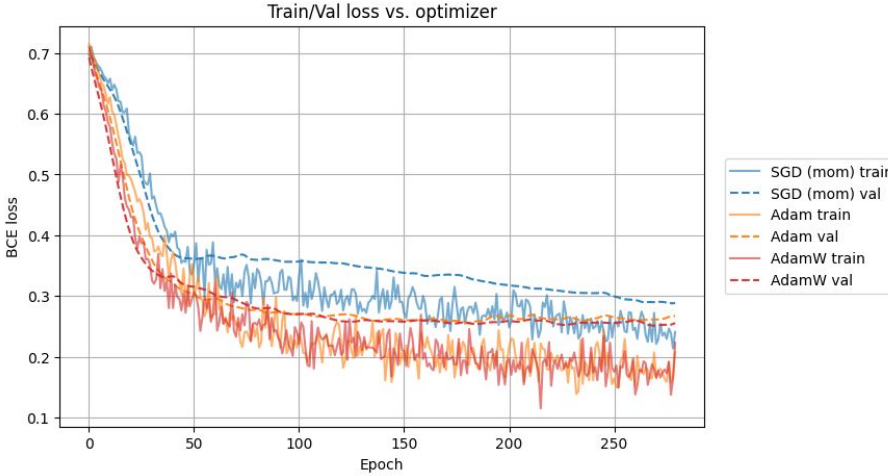


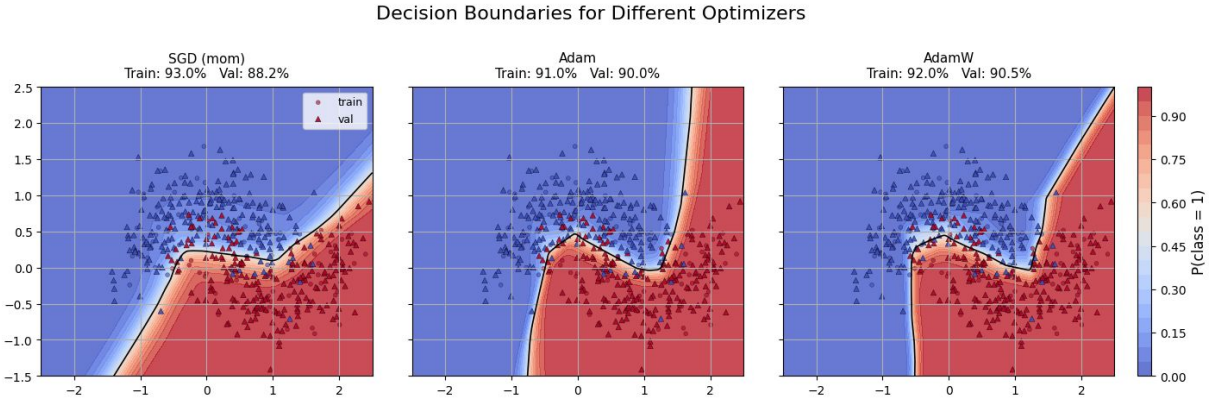
Hyperparameters (optimizers)

- More detail in this article
- **Optimizers control how gradients are transformed into parameter updates during training**

Examples:

- Stochastic Gradient Descent (SGD) with momentum
  - Simple, stable updates
  - Often performs well on unseen data, not just the training set
  - Sensitive to learning rate and scaling
- Adaptive Moment Estimation (Adam)
  - Adaptive learning rates per parameter
  - Fast convergence, less sensitive to hyperparameter choice
  - Can overfit or generalize worse than SGD
- AdamW
  - Adam with decoupled weight decay
  - More reliable regularization behavior
  - Standard choice for modern deep models

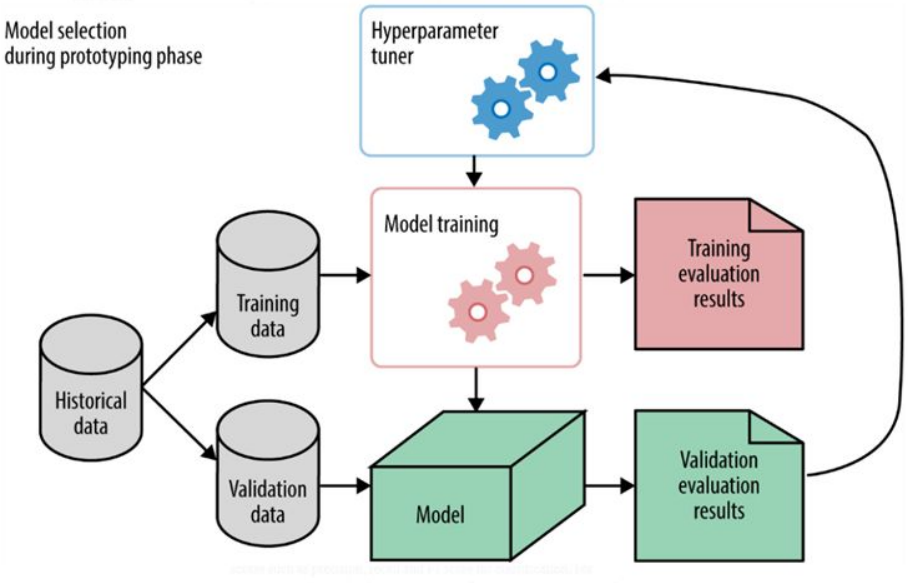




Optuna

What is Optuna?

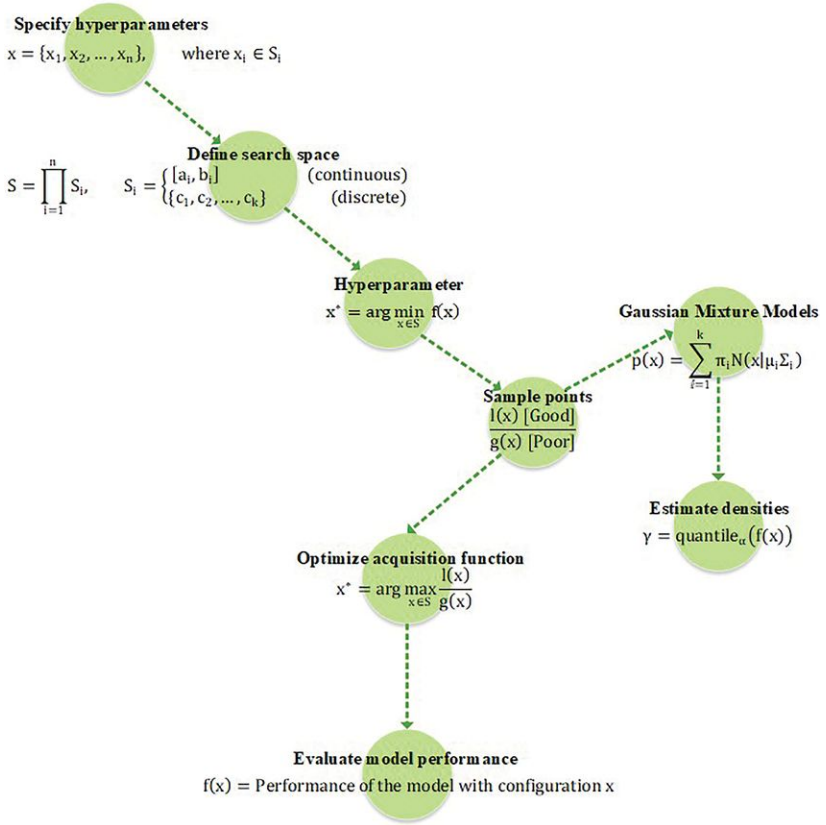
- Many hyper parameters means manual tuning is bad
  - Too slow
  - Suboptimal tuning means you spend more resources (time, computing power, etc.) training unused models
- Optuna is a python package that solves this problem
  - Framework for optimization black box objective functions
  - Technically not an ML package, but rather a package that supports many optimized sampling strategies



Example Optuna workflow diagram

How Optuna Works

- Optuna supports many sampling algorithms, examples:
  - Grid search
  - Random search
  - Gaussian process-based Bayesian optimization
- For single object functions, the default for Optuna is Tree-Structured Parzen Estimator (TPE)



TPE flow diagram (in a nutshell)

Optuna in Practice

- In practice, Optuna abstracts away all the details:
  1. Define object, return the loss
  2. Inside the objective, define hyper parameters to optimize with ranges and suggestions for how to optimize
  3. Return the object to minimize (or maximize)
  4. Run the study
- Optuna will handle storing the history, optimizing the search, etc.

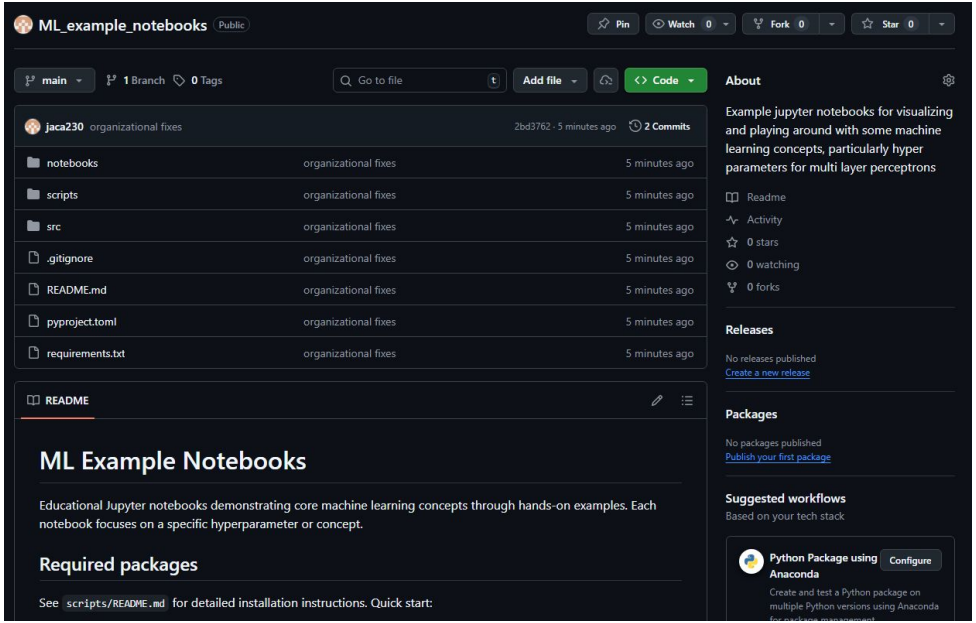
```python
def objective(trial):
    lr = trial.suggest_loguniform("lr", 1e-5, 1e-2)
    hidden = trial.suggest_int("hidden_dim", 32, 256)
    dropout = trial.suggest_float("dropout", 0.0, 0.5)

    model = Model(hidden_dim=hidden, dropout=dropout)
    optimizer = AdamW(model.parameters(), lr=lr)

    val_loss = train_and_validate(model, optimizer)
    return val_loss

study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=100)

```

Example code to run an optuna study

Auxiliary Slides

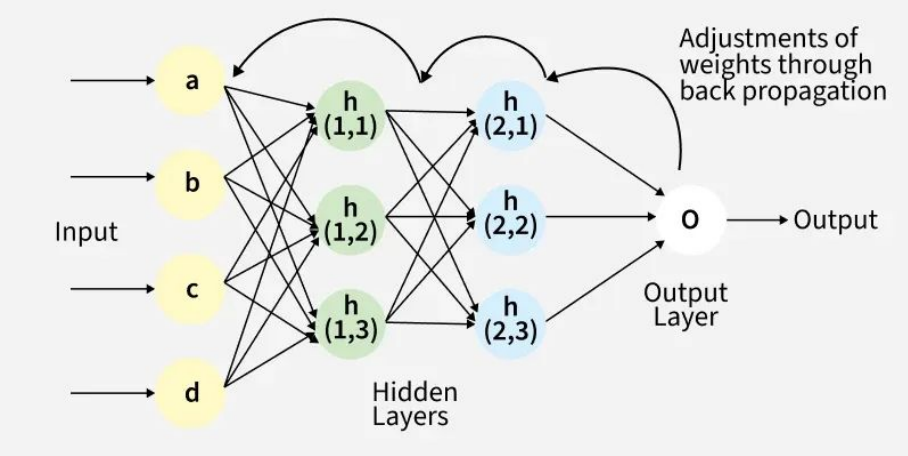
Example Notebooks

- Most plots in this presentation were generated using some interact jupyter notebooks I made
- In these notebooks, one can change parameters to “play around” and better visualize the effects of each hyperparameter
- See [github repo]



Backpropagation (Overview)

- Much more detail in this article, and wikipedia
- **Backpropagation computes gradients of the loss with respect to all model parameters by applying the chain rule backward through the network**
- Compute the gradient of the loss in the space of weights
    - Optimal weight updates for the next iteration



Backpropagation (Explicit, Part I)

- Diagram with some labels to conceptualize the variables

Indices:

$l \equiv$ layer index

$i \equiv$ neuron index in layer l

$j \equiv$ neuron index in layer $l-1$

$k \equiv$ neuron index in layer $l+1$

$t \equiv$ training iteration

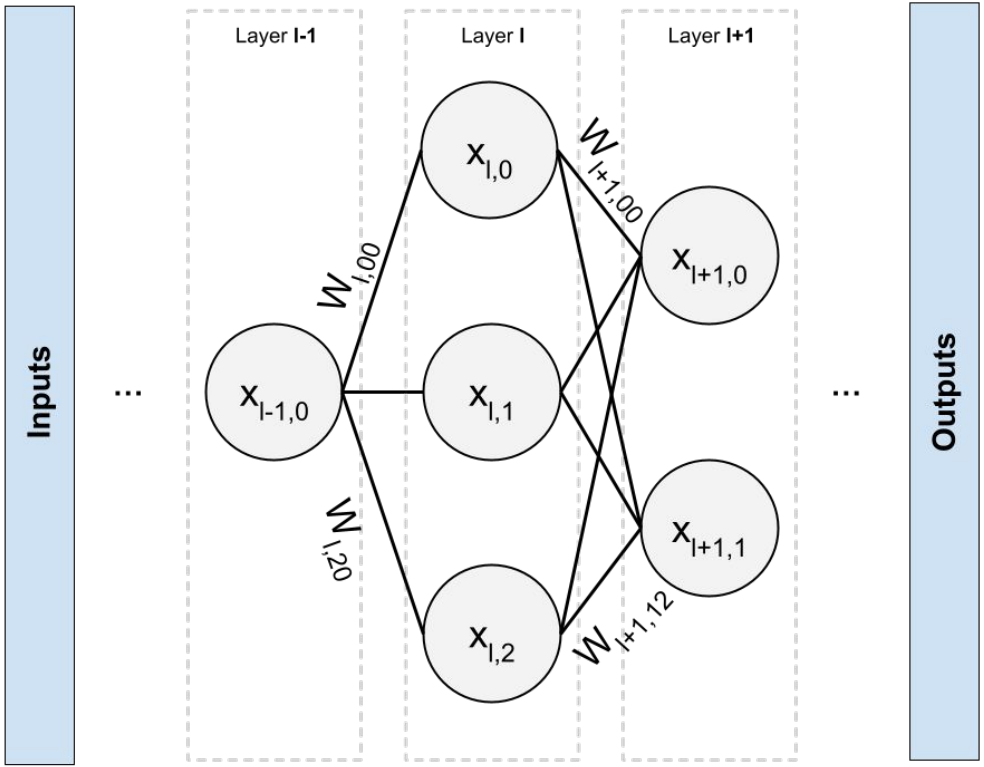
Core quantities:

$L \equiv$ loss (scalar)

$x_{l,i} \equiv$ activation of neuron i in layer l

$W_{l,ij} \equiv$ weight from $(l-1, j)$ to (l, i)

$\alpha \equiv$ learning rate



Backpropagation (Explicit, Part II)

- Equations for how to update weights from back propagation

Indices:

$l \equiv$ layer index
$i \equiv$ neuron index in layer l
$j \equiv$ neuron index in layer $l-1$
$k \equiv$ neuron index in layer $l+1$
$t \equiv$ training iteration

Core quantities:

$L \equiv$ loss (scalar)
$x_{l,i} \equiv$ activation of neuron i in layer l
$W_{l,ij} \equiv$ weight from $(l-1, j)$ to (l, i)
$\alpha \equiv$ learning rate

1. Backpropagate activation gradients:

$$\frac{\partial L}{\partial x_{l,i}} = \sum_k \frac{\partial L}{\partial x_{l+1,k}} \frac{\partial x_{l+1,k}}{\partial x_{l,i}}$$

2. Weight gradient via local chain rule:

$$\frac{\partial L}{\partial W_{l,ij}} = \frac{\partial L}{\partial x_{l,i}} \frac{\partial x_{l,i}}{\partial W_{l,ij}}$$

3. Gradient descent update to weights:

$$W_{l,ij}^{(t+1)} = W_{l,ij}^{(t)} - \alpha \frac{\partial L}{\partial W_{l,ij}}\bigg|_{W=W^{(t)}}$$

Backpropagation (Explicit, Part III)

- The back propagation step is a recursive step

Indices:

$l \equiv$ layer index
$i \equiv$ neuron index in layer l
$j \equiv$ neuron index in layer $l-1$
$k \equiv$ neuron index in layer $l+1$
$t \equiv$ training iteration

Core quantities:

$L \equiv$ loss (scalar)
$x_{l,i} \equiv$ activation of neuron i in layer l
$W_{l,ij} \equiv$ weight from $(l-1,j)$ to (l,i)
$\alpha \equiv$ learning rate

The backpropagation step:

$$\frac{\partial L}{\partial x_{l,i}} = \sum_k \frac{\partial L}{\partial x_{l+1,k}} \frac{\partial x_{l+1,k}}{\partial x_{l,i}}$$

This formula is recursive, so for N layers you have explicitly:

$$\frac{\partial L}{\partial x_{0,j}} = \sum_{i_1} \sum_{i_2} \cdots \sum_{i_N} \frac{\partial L}{\partial x_{N,i_N}} \prod_{m=0}^{N-1} \frac{\partial x_{m+1,i_{m+1}}}{\partial x_{m,i_m}}$$

Or more commonly you can write in Jacobian form, which cleans up the notation

$$\frac{\partial L}{\partial x_0} = \prod_{m=0}^{N-1} \left(\frac{\partial x_{m+1}}{\partial x_m} \right)^\top \frac{\partial L}{\partial x_N}$$

Backpropagation (Explicit, Part IV)

- The actual values of $x_{l,i}$ are *architecture dependent*
- In this simple example, you can compute them by defining a bias and activation function (eg. softmax)

We can define $z_{l,i}$ for the raw computed "response" from the previous layer:

$$z_{l+1,k} = \sum_j W_{l+1,kj}\, x_{l,j} + b_{l+1,k}$$

Then we apply some activation, usually to constrain values between 0 and 1 to prevent exponential blowup. This allows to (finally) formally compute all needed partial derivatives:

$$x_{l+1,k} = \phi(z_{l+1,k})$$

$$\frac{\partial x_{l+1,k}}{\partial x_{l,i}} = \phi'(z_{l+1,k})\, W_{l+1,ki}$$

Core quantities:

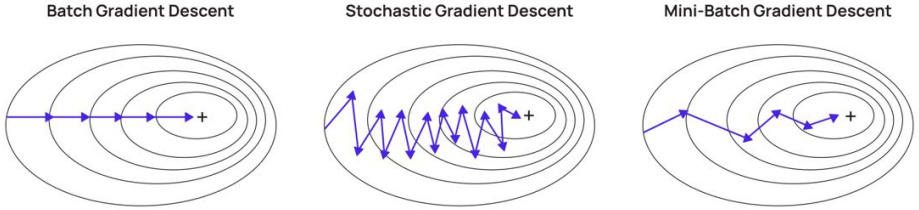
$L \equiv$ loss (scalar)
$x_{l,i} \equiv$ activation of neuron i in layer l
$W_{l,ij} \equiv$ weight from $(l-1, j)$ to (l, i)
$\alpha \equiv$ learning rate
$z_{l,i} \equiv$ pre-activation (raw linear response)
$b_{l,i} \equiv$ bias parameter for neuron (l, i)
$\phi(\cdot) \equiv$ activation function

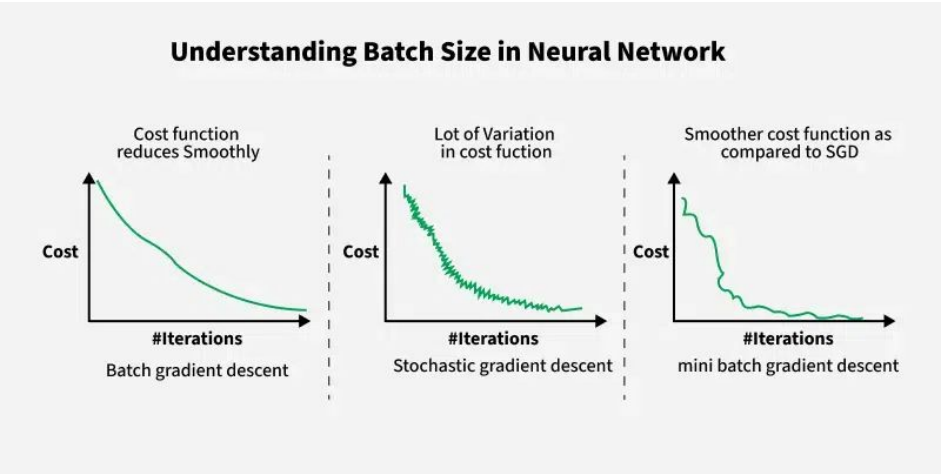
Indices:

$l \equiv$ layer index
$i \equiv$ neuron index in layer l
$j \equiv$ neuron index in layer $l-1$
$k \equiv$ neuron index in layer $l+1$
$t \equiv$ training iteration

Hyperparameters (Gradient Descents)

- Much more detail in this article
- Batch Gradient Descent (BGD):
  - Uses the entire dataset in one iteration. The batch size is equal to the total number of training samples.
- Mini-Batch Gradient Descent (MBGD):
  - Uses a predefined number of samples from the dataset for each update. This method lies between batch and stochastic gradient descent (SGD).
- Stochastic Gradient Descent (SGD):
  - Processes only one sample at a time for each update, making the batch size equal to 1.





Hyperparameters (why are more layers harder to train)

- More layers → more loss instability
  - Due to back propagation
  - The size of your step becomes more “untrainable” the more layers you add
- Cases:
  - λ ≈ 1
    - Gradients stable as model layers increase
  - λ < 1
    - Gradients vanish as model layers increase
    - No more learning occurs!
  - λ > 1
    - Gradients blow up as model layers increase
    - Cannot converge on solution
- One goal of model architecture choice is to get λ ≈ 1

Backprop signal:
$$\frac{\partial L}{\partial x_0} = \prod_{m=0}^{N-1} \left(\frac{\partial x_{m+1}}{\partial x_m} \right)^\top \frac{\partial L}{\partial x_N}$$

Layer Jacobian:
$$x_{m+1} = \phi(W_{m+1} x_m + b_{m+1})$$
$$\frac{\partial x_{m+1}}{\partial x_m} = D_{m+1} W_{m+1}$$
$$\text{where} \quad D_{m+1} = \text{diag}(\phi'(W_{m+1} x_m + b_{m+1}))$$

Substitute:
$$\frac{\partial L}{\partial x_0} = \left(\prod_{m=0}^{N-1} W_{m+1}^\top D_{m+1} \right) \frac{\partial L}{\partial x_N}$$

Take norms:
$$\left\| \frac{\partial L}{\partial x_0} \right\| \leq \left\| \frac{\partial L}{\partial x_N} \right\| \prod_{m=0}^{N-1} \|W_{m+1}\| \, \|D_{m+1}\|$$

$$\implies E\left[\left\| \frac{\partial L}{\partial x_0} \right\| \right] \approx \left\| \frac{\partial L}{\partial x_N} \right\| \left(E[\|W\|] \, E[\|D\|] \right)^N$$

$$\text{Let} \quad \lambda \equiv E[\|W\|] \, E[\|D\|]$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part I)

- Given a (possibly stochastic) black box function you want to minimize (ex. model loss)

$$y = f(\theta)$$

- Hyperparams ≡ θ
- Define the following:

$$\gamma \equiv \text{"good"/"bad" fraction (say, 0.2)}$$
$$y^* \text{ is chosen such that } P(y < y^*) = \gamma$$
$$\text{so } y^* \text{ is in, say, the best } 20\% \text{ of losses}$$
$$\max(y^* - y, 0) \equiv \text{improvment}$$

- Then, for any given theta we can define *expected improvement*

$$\mathrm{EI}(\theta) \equiv \mathbb{E}[\max(y^* - y, 0) \mid \theta] = \int_{-\infty}^{y^*} (y^* - y)\, p(y \mid \theta)\, dy$$

- Goal: Maximize expected improvement

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part II)

- For a 1 dimensional y, it turns out to be easier to work with $p(\theta \mid y)$, we can invert using Bayes' rule:

$$p(y \mid \theta) = \frac{p(\theta \mid y)\, p(y)}{p(\theta)}$$

- And substitute

$$\mathrm{EI}(\theta) \propto \int_{-\infty}^{y^*} (y^* - y)\, \frac{p(\theta \mid y)}{p(\theta)}\, p(y)\, dy$$

- Since we don't know every value of y this becomes an impossible task. We must make an approximation by dividing into "good" and "bad" distributions

$$p(\theta \mid y) \approx \begin{cases} p(\theta \mid y < y^*) \equiv \ell(\theta), & y < y^* \text{ (good)} \\ p(\theta \mid y \geq y^*) \equiv g(\theta), & y \geq y^* \text{ (bad)} \end{cases}$$

$$p(\theta) \approx \ell(\theta) \int_{y<y^*} p(y)\, dy + g(\theta) \int_{y \geq y^*} p(y)\, dy = \gamma l(\theta) + (1-\gamma) g(\theta)$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part III)

- The integral, by construction, only cares about the “good” region, so the integral simplifies to

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{p(\theta)} \int_{-\infty}^{y^*} (y^* - y)\, p(y)\, dy$$

- But the integral is now constant in theta! So we have:

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{p(\theta)}$$

- Where $p(\theta) = \gamma\, \ell(\theta) + (1 - \gamma)\, g(\theta)$ so we can write:

$$\mathrm{EI}(\theta) \propto \frac{\ell(\theta)}{\gamma\, \ell(\theta) + (1 - \gamma)\, g(\theta)}$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part IV)

- But γ is fixed, so

$$\begin{aligned}\arg\max_{\theta} \mathrm{EI}(\theta) &= \arg\max_{\theta} \frac{\ell(\theta)}{\gamma\,\ell(\theta) + (1-\gamma)\,g(\theta)} \\ &= \arg\max_{\theta} \frac{\ell(\theta)/g(\theta)}{\gamma\,\ell(\theta)/g(\theta) + (1-\gamma)} \\ &= \arg\max_{\theta} \frac{\ell(\theta)}{g(\theta)}\end{aligned}$$

- Where the final step is because x/(bx+c) is monotonic in x for x >0 ,let x = l/g

- In other words, we just need to find $\arg\max_{\theta} \frac{\ell(\theta)}{g(\theta)}$ which is doable via algorithm!

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part V)

- Now to define the algorithm, first we observe T (~10) trials randomly:

$$\textbf{Observed trials:} \quad \mathcal{D} = \{(\theta^{(i)}, y^{(i)})\}_{i=1}^{T}$$
$$\textbf{Quantile threshold:} \quad P(y < y^*) = \gamma$$
$$\mathcal{D}_{\text{good}} = \{\theta^{(i)} : y^{(i)} < y^*\}$$
$$\mathcal{D}_{\text{bad}} = \{\theta^{(i)} : y^{(i)} \geq y^*\}$$

- From this data, we want to build:

$$\ell(\theta) \equiv p(\theta \mid y < y^*)$$
$$g(\theta) \equiv p(\theta \mid y \geq y^*)$$

- So we define

$$\theta \equiv (\theta_1, \theta_2, \ldots, \theta_d)$$

- And assume:

$$\textbf{Independence assumption:} \quad p(\theta \mid y < y^*) \approx \prod_{k=1}^{d} p(\theta_k \mid y < y^*)$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part VI)

- This allows us to write:

$$\ell(\theta) \approx \prod_{k=1}^{d} \hat{p}_{\text{good}}(\theta_k)$$

$$g(\theta) \approx \prod_{k=1}^{d} \hat{p}_{\text{bad}}(\theta_k)$$

- We can fit to our samples to get a continuous distribution spaces for each param

$$\hat{p}_{\text{good}}(\theta_k) \leftarrow \text{fit to } \{\theta_k^{(i)} : \theta^{(i)} \in \mathcal{D}_{\text{good}}\}$$

$$\hat{p}_{\text{bad}}(\theta_k) \leftarrow \text{fit to } \{\theta_k^{(i)} : \theta^{(i)} \in \mathcal{D}_{\text{bad}}\}$$

- Then we sample from the “good” spaces for M candidates index by m

$$\tilde{\theta}_{m,k} \sim \hat{p}_{\text{good}}(\theta_k), \qquad k = 1, \ldots, d$$

- And finally, choose our next theta, add it to the data set, and repeat

$$\theta^{(t)} = \arg\max_{\tilde{\theta}_m} \frac{\ell(\tilde{\theta}_m)}{g(\tilde{\theta}_m)} = \prod_{k=1}^{d} \frac{\hat{p}_{\text{good}}(\tilde{\theta}_{m,k})}{\hat{p}_{\text{bad}}(\tilde{\theta}_{m,k})}$$

Note: See paper on arxiv

What is a Tree-Structured Parzen Estimator (TPE) (Part VII)

- How do we obtain our fits from the data?
- Uses Kernel Density Estimation (KDE)
  - The actual fit is done when determining each “gaussian kernel” (or sigma)
  - Likelihood-optimal is expensive ~O(n^2d)
    - n = # samples
    - d = # hyperparameters
  - Optuna uses the “heuristic” version, which requires choosing some value for c.

Given samples: $\{\theta_k^{(i)}\}_{i=1}^n, \qquad \theta_k^{(i)} \in \mathbb{R}$

Kernel density estimate:

$$\hat{p}(\theta_k) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{K}\left(\frac{\theta_k - \theta_k^{(i)}}{h_k}\right)$$

Gaussian kernel:

$$\mathcal{K}(u) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{u^2}{2}\right)$$

Bandwidth (likelihood-optimal):

$$h_k^\star = \arg\max_h \sum_{i=1}^{n} \log \hat{p}_{-i}(\theta_k^{(i)} \mid h)$$

Bandwidth (heuristic used in practice):

$$h_k = c \cdot \mathrm{std}\left(\{\theta_k^{(i)}\}_{i=1}^n\right), \qquad c > 0$$