Deep Dive into Overfitting and Regularization

Farid Afzali, Ph.D., P.Eng.

Deep Learning metaparameters

- Model Architecture
- Number of hidden layers
- Number of units per layer
- Cross validation size
- Mini-batch size
- Activation functions
- Optimization functions
- Learning rate
- Dropout
- Loss function
- Data Normalization
- Weight normalization
- Weight initialization

Mata Parameters

1. **It is simply impossible to search the entire metaparameter space.**
   - ❑ Searching all possible values of metaparameters in a model is not feasible.
   - ❑ There are too many potential configurations to consider.
2. **It is difficult to know whether you are using the “best” model for your problem.**
   - ❑ Determining the optimal model for a specific task can be challenging.
   - ❑ Different models might perform similarly, making it hard to select one as the "best".
3. **Fortunately, parametric experiments on some metaparameters are feasible.**
   - ❑ While not all metaparameters can be exhaustively tested, some can be explored.
   - ❑ Selective experimentation allows for better tuning without exploring every possible value.
4. **Ultimately, you must use a combination of experience, intuition, previous successes, and empirical exploration.**
   - ❑ A combination of past knowledge, gut feeling, and hands-on experimentation is essential for model tuning.
   - ❑ Sole reliance on any one method might not yield the best results.

Dataset Python Practice

Wine Quality

Donated on 10/6/2009

Two datasets are included, related to red and white vinho verde wine samples, from the north of Portugal. The goal is to model wine quality based on physicochemical tests (see [Cortez et al., 2009], http://www3.dsi.uminho.pt/pcortez/wine/).

Dataset Characteristics	Subject Area	Associated Tasks
Multivariate	Business	Classification, Regression

Feature Type	# Instances	# Features
Real	4898	11

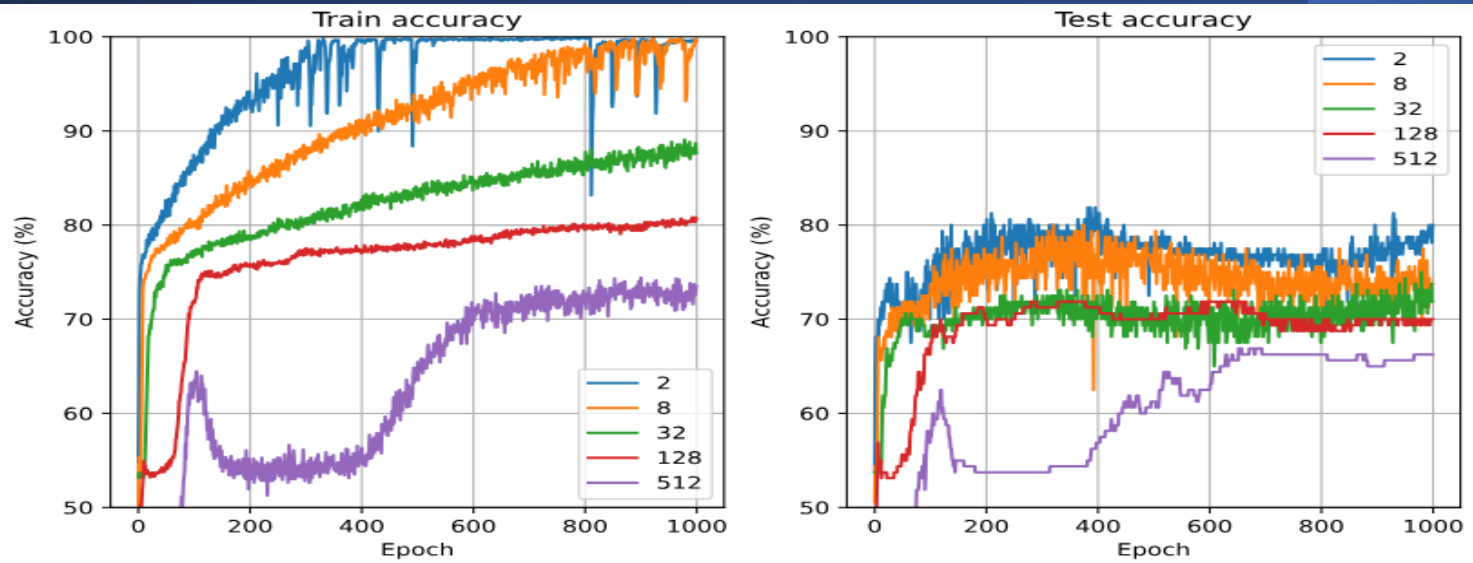
Dataset Information

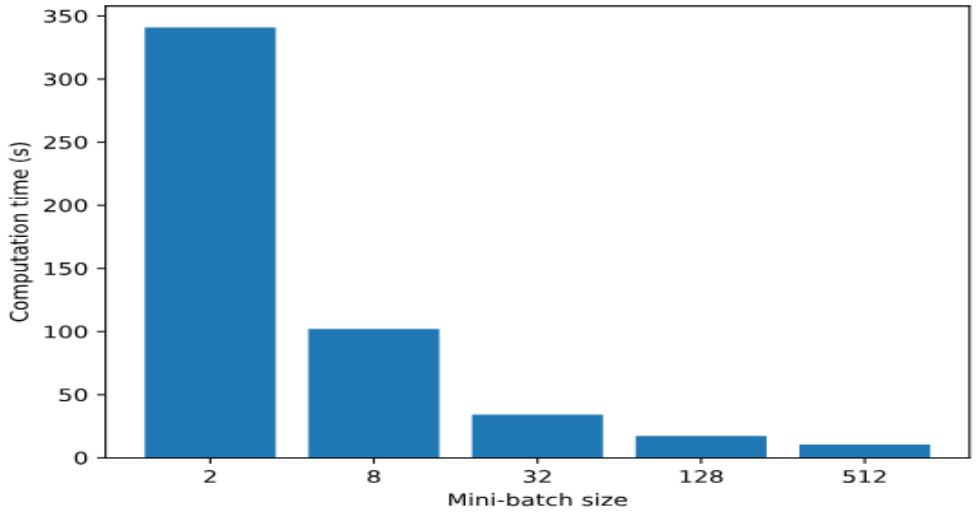
Additional Information

The two datasets are related to red and white variants of the Portuguese "Vinho Verde" wine. For more details, consult: http://www.vinhoverde.pt/en/ or the reference [Cortez et al., 2009]. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).

These datasets can be viewed as classification or regression tasks. The classes are ordered and not balanced (e.g. there are many more normal wines than excellent or poor ones). Outlier detection algorithms could be used to detect the few excellent or poor wines. Also, we are not sure if all input variables are relevant. So it could be interesting to test feature selection methods.

Computational Time for different numbers of batches, PP#2





Feature Scaling

- Feature scaling is an essential preprocessing step in many machine learning workflows, and it offers several benefits to the modeling process. It involves transforming the features (variables) in your dataset to bring them into a similar scale or range. The main ideas behind feature scaling are as follows:

1. **Equalizes the Influence of Features:** Feature scaling ensures that all features contribute equally to the modeling process. When features have different scales or units, those with larger magnitudes can dominate the learning process and have a disproportionate influence on the model's decisions. Scaling mitigates this issue, allowing each feature to contribute more evenly.

2. **Improves Convergence:** In iterative optimization algorithms, such as gradient descent, feature scaling can help the algorithm converge faster and more reliably. When features have vastly different scales, the optimization process can take longer to find the optimal solution or may become unstable. Scaling helps the optimization process by providing a more balanced landscape.

3. **Enhances Interpretability:** Scaling makes it easier to interpret the importance of coefficients or feature contributions in models like linear regression. With scaled features, the magnitude of the coefficients directly reflects their impact on the target variable, and it's easier to compare their importance.

4. **Facilitates Distance-Based Metrics:** Machine learning models that rely on distance metrics, such as k-Nearest Neighbors (KNN), are particularly sensitive to feature scaling. Without scaling, features with larger ranges or magnitudes can disproportionately affect the distance calculations, potentially leading to suboptimal results. Scaling ensures that all features have a similar influence on distance-based computations.

Feature Scaling Benefits

- **Great Increases in Performance:** Feature scaling can indeed lead to significant improvements in model performance. By ensuring that features are on a similar scale and reducing the influence of outliers, scaling can help models converge faster and make more accurate predictions.

- **Necessary for Some Models:** Feature scaling is essential for certain machine learning models, particularly those that rely on distance metrics like K-Nearest Neighbors (KNN) and clustering algorithms like K-Means. These models can perform poorly without feature scaling, as they are sensitive to the magnitudes of features.

- **Virtually No "Real" Downside:** In general, there are very few downsides to scaling features. While some might argue that feature scaling introduces a slight loss of interpretability (since the scaled features are no longer in their original units), this is a minor concern compared to the benefits gained in terms of model performance and reliability.

Feature Scaling

- Two main ways to scale features:
  - *Standardization*

$$X_{changed} = \frac{X - \mu}{\sigma}$$

    - Rescales data to have a mean (**μ**) of 0 and standard deviation (**σ**) of 1.
  - *Normalization*

$$X_{changed} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

    - Rescales all data values to be between 0-1.

Feature Scaling Scikit-Learn

1..fit() Method: In this step, the method calculates and computes the necessary statistics or parameters from the data that will be used for the transformation. For feature scaling, this typically involves calculating statistics like the minimum (Xmin), maximum (Xmax), mean, and standard deviation of the features.

2..transform() Method: After the necessary statistics are calculated during the fitting step, the .transform() method applies the scaling or transformation to the data. It uses the computed statistics to transform the data according to a specific scaling method (e.g., min-max scaling, standardization) and returns the transformed version of the data.

- This separation of fitting and transforming is a common pattern in many preprocessing techniques and allows you to reuse the same transformation parameters on different datasets or apply the same transformation consistently to training and testing datasets. This approach ensures that the same scaling is applied to both training and testing data, maintaining data consistency.

Feature Scaling Some Important Notes

- ✓ **Fitting to Training Data:** When you perform any preprocessing step, such as calculating statistics for feature scaling, it's essential to fit the transformation only to the training data.
  - ❑ This means that you compute the necessary statistics (e.g., mean, standard deviation, minimum, maximum) using the training dataset alone.
  - ❑ The reason for this is to prevent any knowledge or information about the test data from leaking into the training process.
- ✓ **Avoiding Data Leakage:** Using the full dataset (both training and testing) for fitting can lead to data leakage.
  - ❑ Data leakage occurs when information from the test set inadvertently influences the modeling process.
  - ❑ This can result in overly optimistic model performance estimates during training, leading to poor generalization when the model encounters unseen data.
- ✓ **Maintaining Independence:** The test set should represent completely independent and unseen data that the model has not been exposed to during training.
  - ❑ By fitting preprocessing steps only to the training data, you ensure that the model doesn't rely on any information specific to the test set, which preserves the integrity of your model evaluation.

Feature Scaling-Target Variable??

- In most cases, it is not necessary or advised to scale the label (target variable) in regression problems. Scaling the label can introduce several complications and is generally not a common practice for the following reasons:

1. **Altering the Definition of the Target:** Scaling the label changes the meaning of the target variable. For example, if you scale a house price label, it will no longer represent the actual price of houses but rather a scaled version of it. This makes interpretation and communication of results more challenging.

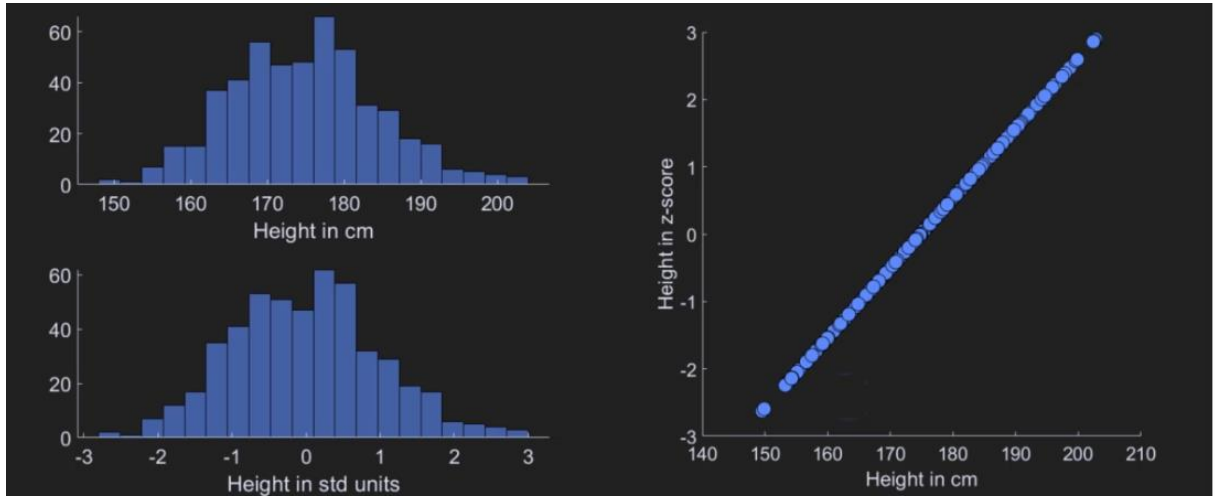
2. **Predicting a Different Distribution:** When you scale the target, you're effectively transforming it to a different distribution. This can lead to confusion when comparing model predictions to the original, unscaled target values.

3. **Loss of Interpretability:** Scaling the target makes it more difficult to interpret the model's coefficients. In linear regression, for instance, the coefficients represent the change in the target variable for a one-unit change in the corresponding feature. Scaling the target would mean that the coefficients no longer have their original interpretation.

4. **Evaluation Metrics:** Common evaluation metrics for regression, such as Mean Absolute Error (MAE) and Mean Squared Error (MSE), are designed to work with unscaled targets. If you scale the target, you would need to use different metrics that account for the scaling to assess model performance accurately.

Data Normalization pp#3

- All samples are processed the same
- All features are treated the same
- Weights remain numerically stable
- Z transform shifts and stretches, but doesn’t change the shape
- Min_max scaling is common for images and uniform-data
- Z_scoring is common for data that are normally distributed.



Batch Normalization

1. **Normalization**: For each mini-batch of data during training, batch normalization calculates the mean and standard deviation of the activations within that batch.
2. **Normalization Operation**: It then normalizes the activations in the batch by subtracting the mean and dividing by the standard deviation. This centers the activations around zero and scales them to have a standard deviation of one.
3. **Scaling and Shifting**: After normalization, the activations are scaled and shifted using learnable parameters (gamma and beta). This allows the model to learn the optimal scaling and shifting for each layer.
4. **Backpropagation**: During training, the gradients are backpropagated through the normalization layer, and the learnable parameters (gamma and beta) are updated through gradient descent.

Model Complexity

- **Faster Convergence**: It helps neural networks converge faster during training by reducing the internal covariate shift problem.
- **Regularization**: Batch normalization acts as a form of regularization, reducing the risk of overfitting.
- **Stability**: It stabilizes the training process, making it less sensitive to the initialization of weights and learning rate choices.
- **Improved Gradient Flow**: Normalized activations can lead to better gradient flow, allowing for the training of deeper networks.

Batch normalization pp#4

$$\widetilde{\mathbf{x}} = \gamma \mathbf{x} + \beta$$

- Batch normalization should only be applied during the training
- It should be off during validation/test, because batch size could different

Where do we need to apply the batch normalization?

Class Activity pp#5

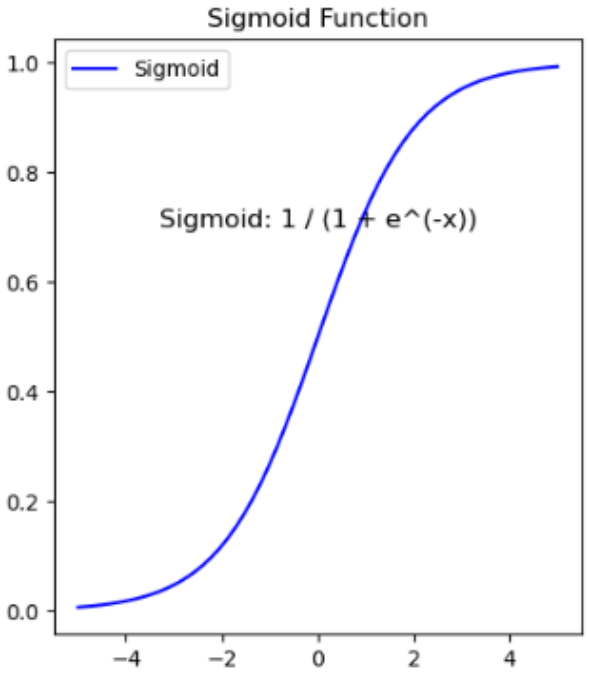
- Go to the code **ANN-2-Multilayer.ipyn**b, add the batch normalization to the code, and see what the results would compare to the previous practice

Activation Function

- Hidden layer
- Nonlinearity
- Compositionality Simple
- Avoid limited range
- Avoid vanishing/exploding gradient

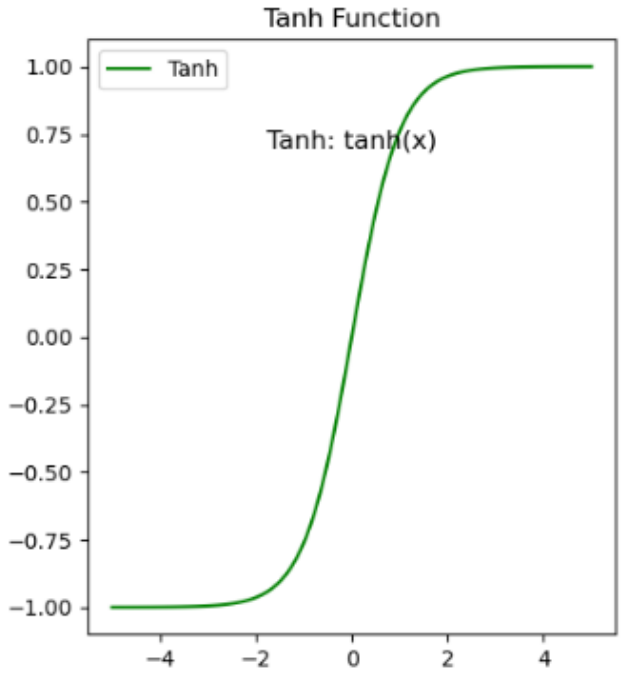
- Output layer
- Nonlinear (classify) or linear (Regression)
- Boundaries of saturation that can be converted to the probabilities [0,1]

Activation Functions



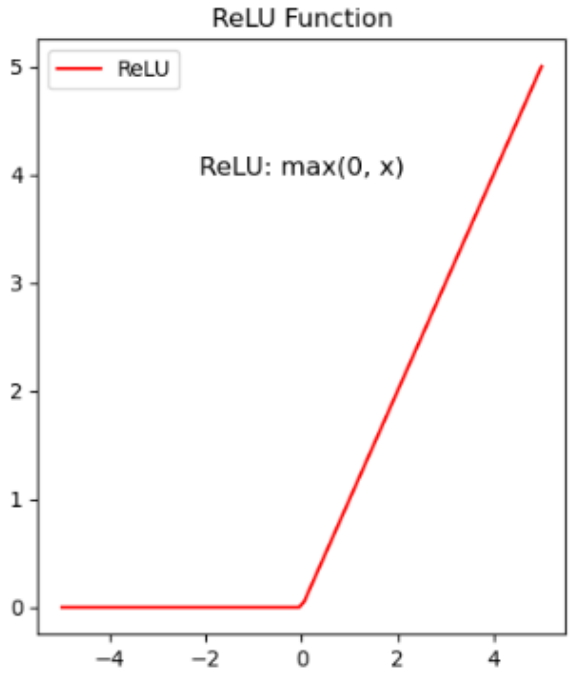
- Nearly Linear
- Limited in [0,1]
- Biased average
- Great for output

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



- Nearly Linear in range
- Limited in [-1,1]
- Unbiased average
- Okay for Hidden

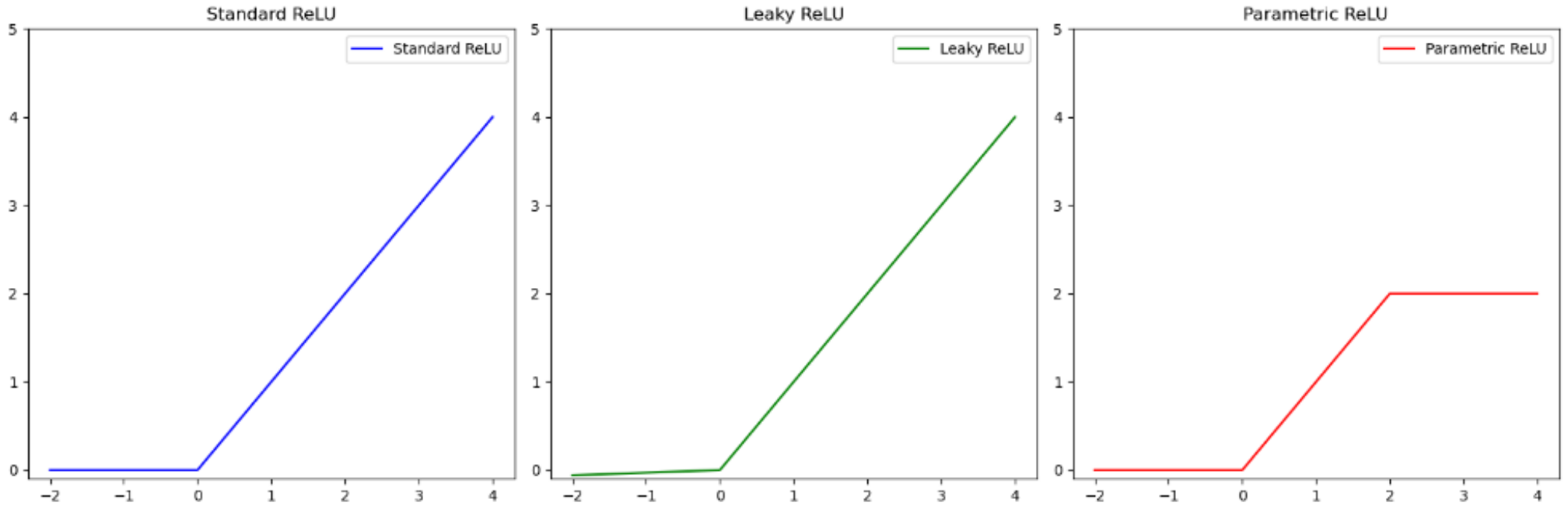
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



- Strongly Nonlinear
- Not limited
- Biased average
- Great for hidden

$$\mathrm{ReLU}(x) = \max(0, x)$$

Activation Functions(pp#6,7,8)



- Strongly Nonlinear
- Not limited
- Biased average
- Great for hidden

$$ReLU : \max(0, x)$$

- Strongly Nonlinear
- Not limited
- Biased average
- a=0.1

$$LeakyReLU : \begin{cases} x & \text{for } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$$

- Strongly Nonlinear
- Saturates
- Avoid blowing up
- N=6

$$ParametricReLU : \begin{cases} N & \text{for } N \leq x \\ x & \text{for } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

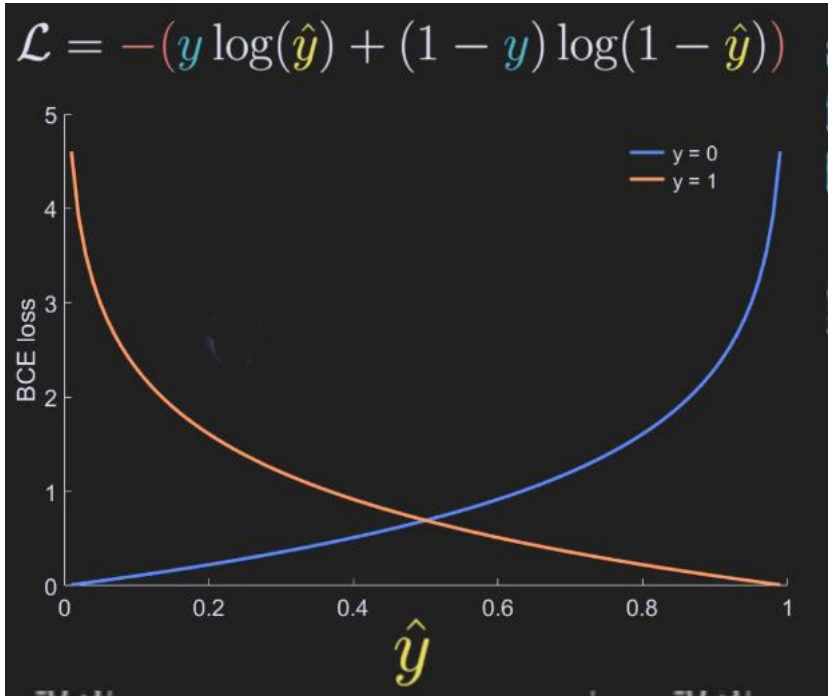
Loss Functions

- Mean Squared error (MSE)
- Use for the continuous data
- When the output is a numerical prediction
- Height, house price, temperature

- Cross Entropy (Logistic)
- Use for the categorical data when the output is probability
- Presence of the disease, animal in picture, text sentiment

Cross Entropy Loss Functions

- Binary Cross Entropy
- Use for two possible answer

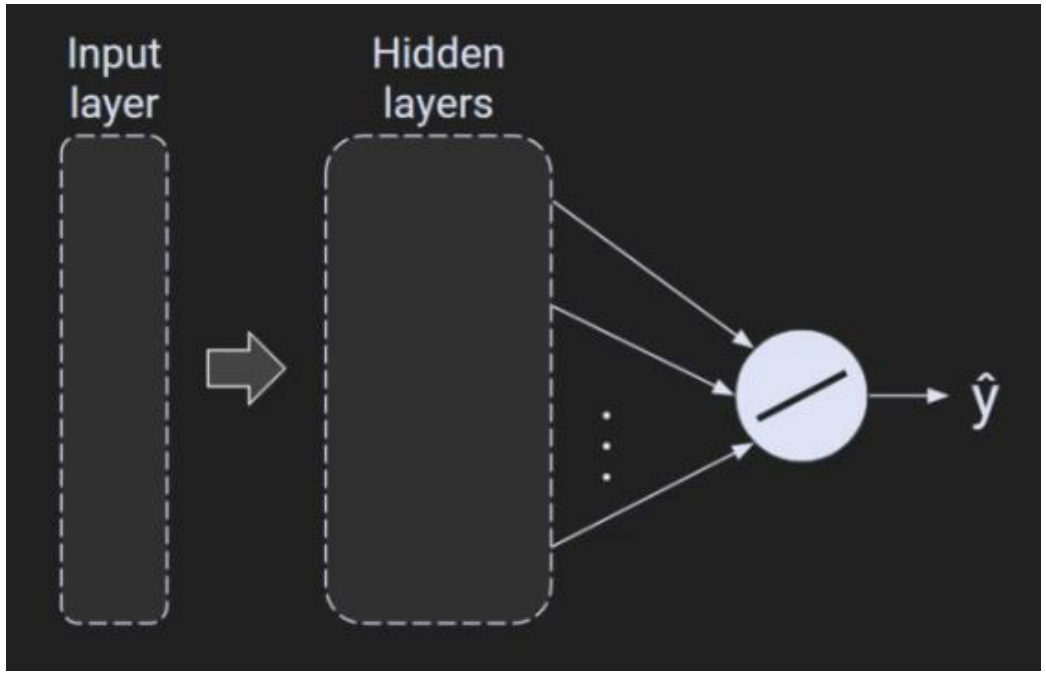


- Categorical Cross Entropy
- Aka negative log-likelihood
- Use with on-hot encoding

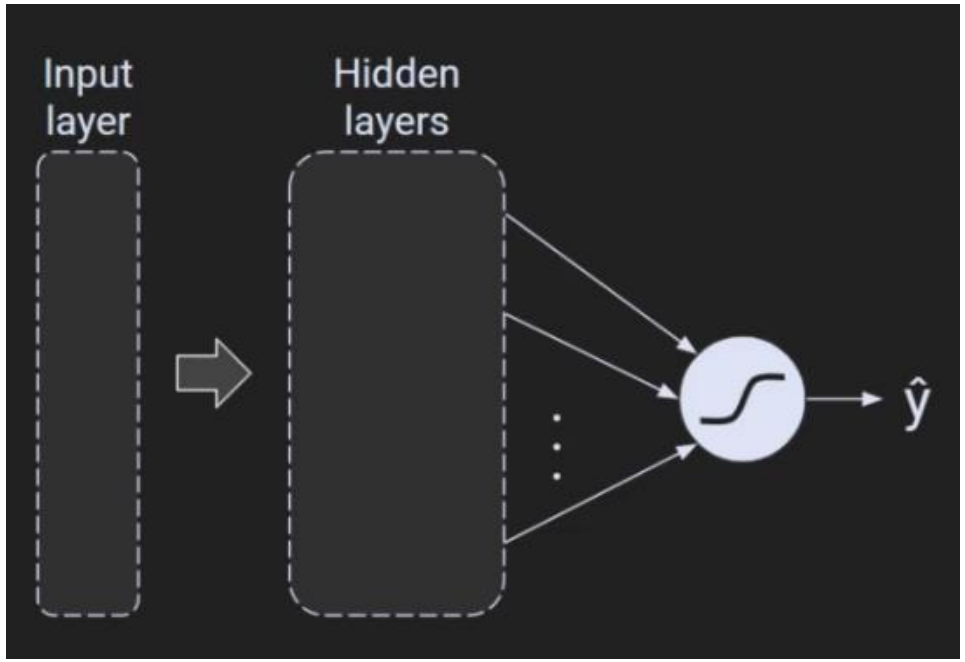
$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{k=1}^{C} y_i^{(k)} \log(\hat{y}_i^{(k)})$$

Output layer architectures

- Prediction of continuous data
- Output layer is just one unit
- It has linear activation function
- MSE loss function

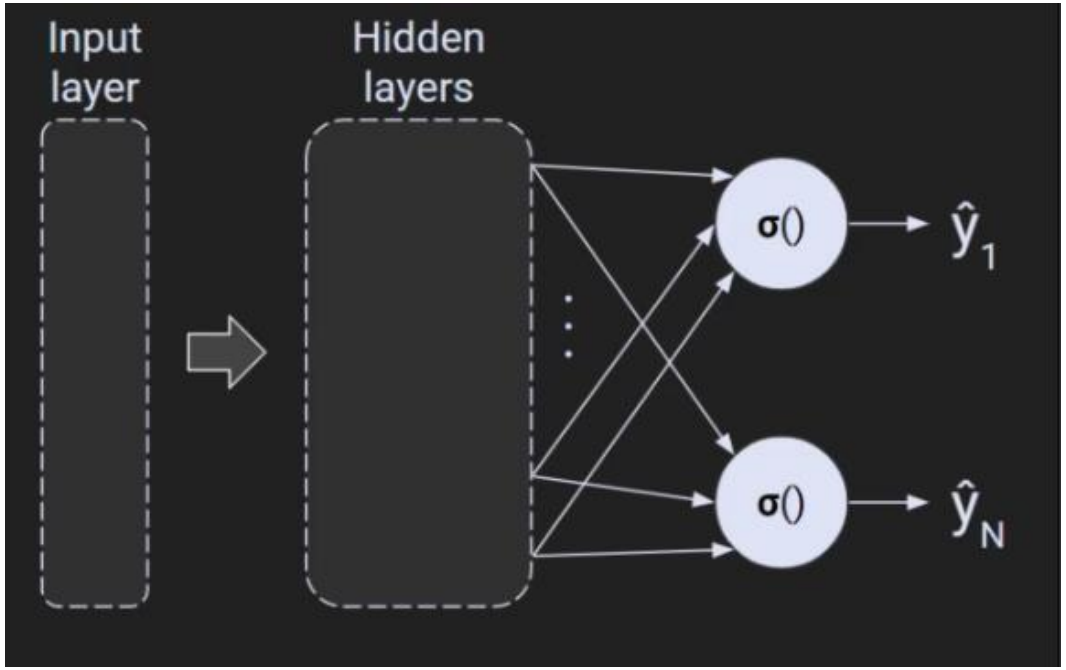


- Binary Classification
- Output layer is one unit
- It has sigmoid activation function
- BCE loss function

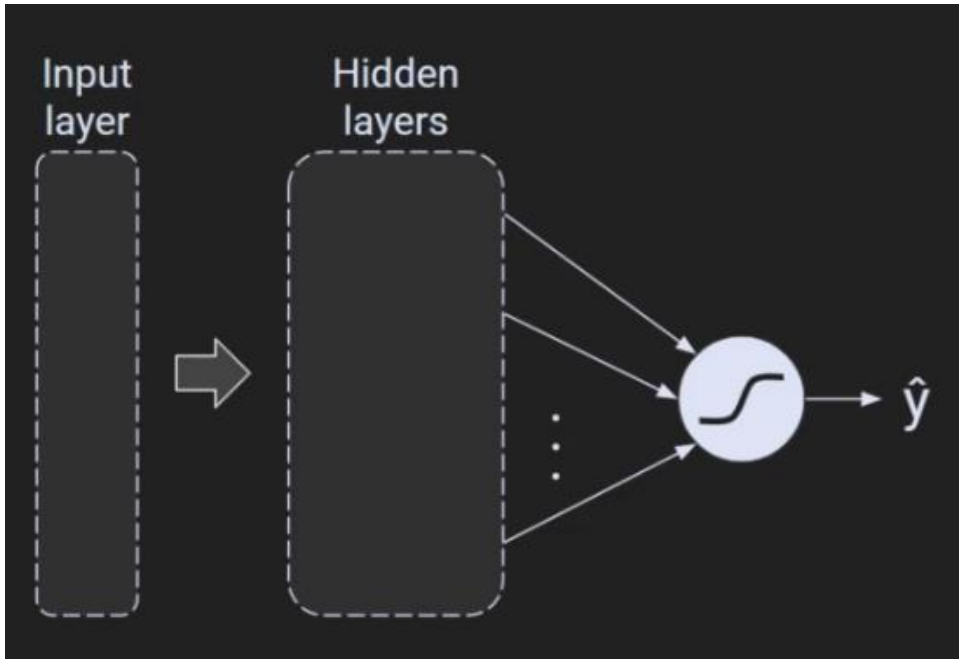


Output layer architectures(pp#9, 10)

- Prediction of continuous data
- Output layer has N unit for N different classes
- It has softmax activation function
- CCE loss function

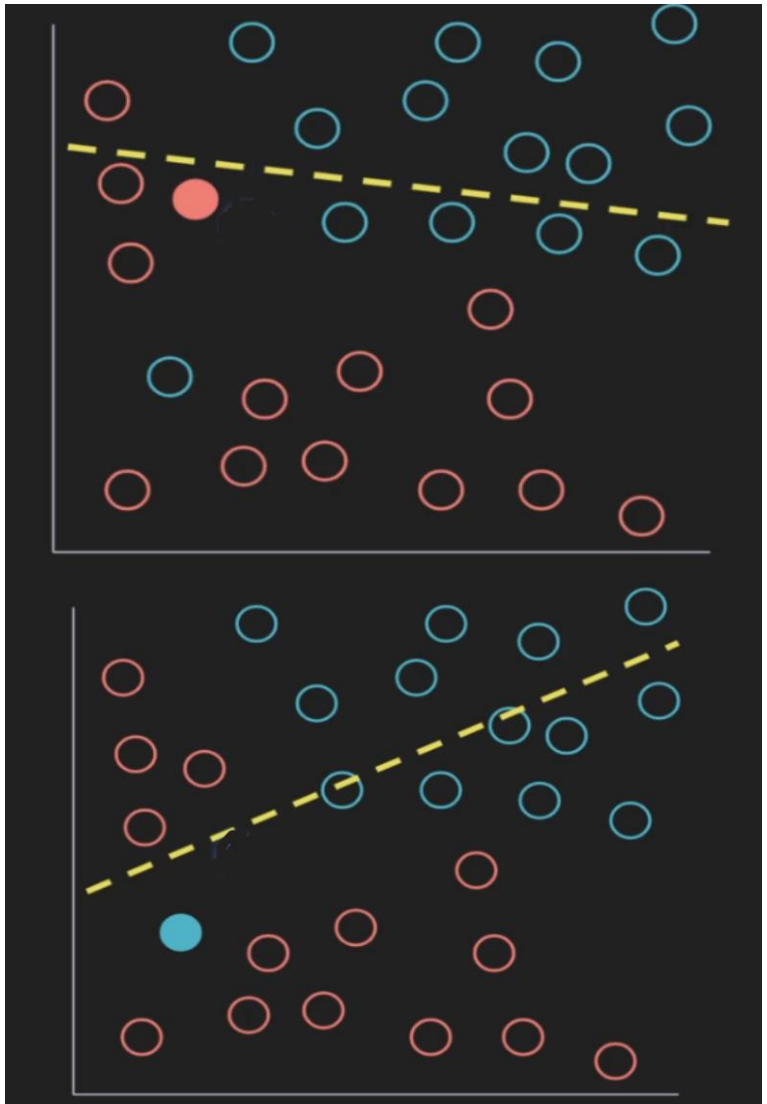


- Binary Classification
- Output layer is one unit
- It has sigmoid activation function
- BCE loss function



Optimizers (minibatch and Momentum)

- SGD: Change the weights after each sample
- This is great if all the samples are similar to each other.
- SGD is very sensitive and can lead to the volatile changes based on the non representative samples
- Mini-batch SGD changes the weights after N sample and averages losses across the N samples

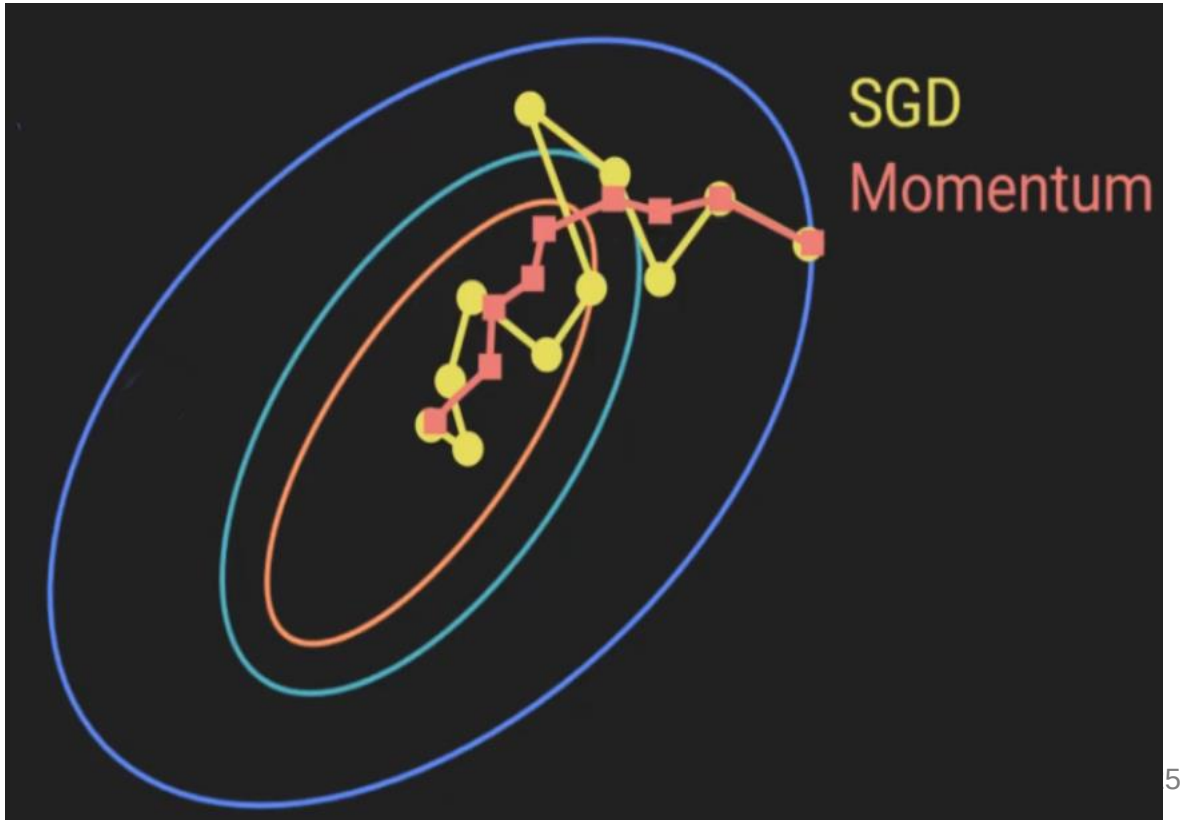


Optimizers (minibatch and Momentum)PP#11

- Update the current weights according to a weighted average of current and previous costs
- Beta can be 0.9 to 0.99
- Weight trajectory is biased by previous direction.
- The particle builds momentum as it moves along the error surface.

$$\mathbf{v} = (1 - \beta) d\mathcal{J} + \beta \mathbf{v}_{t-1}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \mathbf{v}$$



RMSprop

- RMS=root mean square (prop=propagation)
- Similar concept as momentum, Bias the weight changes using dampened previous gradients
- RMSprop Solution: Instead of biasing the gradient, bias the learning rate according to the magnitude of the gradient.
- The step size for adjusting the weight depends on the history of the gradient magnitudes
- LARGE gradient→ Smaller step
- SMALL gradient→ Larger step

$$\mathbf{s} = (1-\beta)\left(d\mathcal{J}\right)^2 + \beta\mathbf{s}_{t-1}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{\sqrt{\mathbf{s}+\epsilon}} d\mathcal{J}$$

Adam(Adaptive Momentum) PP#12, 13, 14, 15

- Formula looks complicated.
- Idea is simple:
  - Combine momentum and RMSprop

$$\mathbf{v} = (1 - \beta_1) d\mathcal{J} + \beta_1 \mathbf{v}_{t-1}$$

$$\mathbf{s} = (1 - \beta_2) (d\mathcal{J})^2 + \beta_2 \mathbf{s}_{t-1}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{\sqrt{\tilde{\mathbf{s}}} + \epsilon} \tilde{\mathbf{v}}$$

Bias correction factor

$$\tilde{\mathbf{v}} = \frac{\mathbf{v}}{1 - \beta_1^t}$$

$$\tilde{\mathbf{s}} = \frac{\mathbf{s}}{1 - \beta_2^t}$$

Recommended parameter settings

$\eta = .001$

$\beta_1 = .9$

$\beta_2 = .999$

$\epsilon = 10^{-8}$ (1E-8)