# Machine learning breadth

- [ ] **Gradient Descent**
  - fundamental optimization algorithm to
  - minimize loss function to align models predictions to match ground truth.

It iteratively adjusts the models parameters, by moving in the opp direction of the gradient to reduce loss, until it converges to the min.

---

- [ ] **learning rate**
  - is the crucial parameter that determines the size of
  - the steps the algorithm takes along the gradient when updating the models parameters.

L.R. too ↑ = overshoot the minima , L.R. ↓ = stuck in local minina

---

- [ ] **Choosing a learning rate**:
  - step decay ①
  - exponential decay ②
  - Adaptive learning rate ③

learning rate schedules
① Reduce by a factor
② Gradually reduce expo
③ Adagrad, Adam, RMSprop

---

- [ ] **Adaptive learning rate**: optimization of gradient descent

① **AdaGrad**: Adaptive Gradient Algorithm
- adapts the learning rate to the parameters, ~~performing smaller updates~~
  ↳ designed to give diff LR to diff features based on frequency.

② **RMSprop**
↳ modifies Adagrad by solving rapidly diminishing L.R.
↳ instead of summing up all past squared gradients, RMS uses a moving average so only a certain # of past gradients make an influence on training. Update: RMS prop divides the L.R. by the square root of this moving average which means it adjusts the rate on recent trends in gradient not the whole history.

---

③ **Adam**: Adds on the benefits of AdaGrad & RMSprop and adds the concept of momentum
↳ Adam tracks moving average of gradients themselves not just their squares which will accelerate optimization in the right direction reducing the oscillation w/RMSprop

☐ **Overfitting** : low loss during training, but poor @ predicting new data
- maybe model is too complex
① ↳ learns the detail and noise in the training data
- How it happens: ② Training data is noisy ③ Model trained for too long
- How to fix : Regularization, pruning layers, early stopping, more data
- How to check : Evaluate performance b/w train & validation/test

---

☐ **Underfitting** : Model is too simple to learn the underlying pattern of the data & can't capture the underlying trend of data
How it happens → Model is too simple, features don't capture enough information about the data.
- How to check → poor performance on train & test
- How to fix → increase complexity, add more features, more interactions
→ Decrease regularization, train for longer

**Metrics** : F1-score, accuracy, precision, recall can tell if model is under or overfitting

---

☐ **ML Metrics**: ① Accuracy: $\frac{\text{Measure of correct prediction}}{\text{Total predictions}}$
② Precision & Recall : TP/TP+FP Recall : TP/TP+FN
③ F1 = 2 · P×R / P+R harmonic mean b/w P & R
④ ROC & AUC : ROC curve is a graphical representation of the contrast b/w true positive rates & false positive rates @ various thresholds

AUC : area under the ROC curve & provides aggregate measure of performance across all thresholds
⑤ MAE : Avg of absolute diff b/w predicted & actual values
⑥ MSE : Avg of squares diff b/w pred & actual

☐ **Feature Engineering** : Turning raw data → features | **categorical features** : have one discrete value
Categories like gender, state, product

☐ **Regularization** : Adds a penalty on different parameters of the model
↳ penalty is applied to the loss function of the model to make sure model does not overfit to the training data & is able to generalize to new, unseen data well

① **L1** adds | mag of coeff | & reduces some feature coeff to 0
↳ feature selection

② **L2** adds penalty equal to square of magnitude
↳ prevents any single param from getting too large

③ **Dropout** : for NN's, it randomly drops unit activations in a network for a single gradient step
↳ randomly deactivates neurons during training

Algorithms : linear regression : assumes linear relationship

**Batch size** : Datasets are split into smaller sizes to train the model. Gradient updates are also done in batches as well to reduce the loss

- $\boxed{\text{SGD}}$ → batch size set to 1, each training instance is considered individually, so more noise but faster updates
- Mini Batch GD → subset of instances @ each iteration

☐ **Kmeans clustering** : partition into k distinct clusters & has a centroid

☐ **Dimensionality Reduction** : helps with the curse of dimensionality
PCA

☐ ↓ data, ↑ features : Data augmentation, dimensionality reduction
risk of overfitting, regularization

☐ **Multicolinearity** : high correlation between features and data

- Data is fed into DNN in batches, which can cause vanishing or exploding gradients
↳ inserted after fully connected layers & before non-linearity

- [ ] Overfitting → ① Regularization ② Augment the data ③ Model complexity reduce ④ Dropout ⑤ Batch normalization

- [ ] A/B testing ↳ helps test a new change, 1 treatment & 1 control grp
↳ strategic bet to make the change, it gives the customer a voice
↳ guardrail metrics

- [ ] Batch Normalization and Group Normalization
↳ streamline the training of DNNs & tackle internal covariate shift: variation in distribution of network layer inputs as the network parameters are updating during training

- [ ] BN : standardizes the inputs to a layer for each mini batch
↳ calculates mean & variance for a batch

- [ ] GN : divides the input into groups and normalizes the data w/in each group using group specific mean & variance

- [ ] Batch Inferences vs real time
↓ offline, cached
↳ model in serving layer, handle a request @ a time or in very small batches

- [ ] ReLU, skip connections → vanishing gradient problem

- [ ] Data Drift : Continuous monitoring, Kolmogorov-Smiroff

- [ ] Transformers outperform due to parallel processing

- [ ] RMSnorm : First Batch normalization normalizes the activation of a previous layer per each batch. Layer norm normalizes the inputs across all features for each data sample in a batch
↳ simplifies layer norm by removing the mean subtraction step & normalizing the activations based only on their RMS