Projet Convex Optimization

Subject A: application of optimization techniques to a concrete machine learning problem : On Kaggle Competition : House Prices- Advanced Regression Techniques

Equipe:

SENECHAL Morgan

LOGEROT Jules

COSTA Thomas

Professeur : BENEDETTI Leonard

Module : ADIF83

Table des matières

I. Introduction

L'optimisation est au cœur de nombreux algorithmes de machine learning, jouant un rôle crucial dans l'amélioration de la performance des modèles. Dans ce projet, nous avons entrepris d'explorer l'application de diverses techniques d'optimisation pour résoudre un problème concret de machine learning : la prédiction des prix de vente des maisons en utilisant le jeu de données de la compétition "House Prices- Advanced Regression Techniques" issues de Kaggle. Ce dataset riche en informations offre une opportunité idéale pour examiner comment différentes stratégies d'optimisation peuvent être utilisées pour affiner les modèles de régression et de réseaux de neurones.

L'objectif principal de ce projet est de comparer l'efficacité de différentes techniques d'optimisation dans le contexte spécifique de la régression pour la prédiction de prix immobiliers. À cet effet, nous avons choisi de construire un modèle de réseau de neurones profond basé sur la métrique MSE, adapté aux caractéristiques complexes et variées du dataset "House Prices- Advanced Regression Techniques". Ce choix s'appuie sur l'hypothèse que les réseaux de neurones, avec leur capacité à apprendre des représentations non linéaires et interactives des données, sont particulièrement adaptés à la modélisation des nuances subtiles impliquées dans la détermination des prix immobiliers.

Dans notre démarche, nous avons mis en œuvre plusieurs optimiseurs, notamment Adam, SGD, RMSprop et Nadam, chacun possédant des caractéristiques qui pourraient influencer de manière significative la convergence du modèle et sa performance finale. Nous avons évalué ces optimiseurs sur plusieurs critères : le nombre d'epochs, le taux d'apprentissage, les paramètres spécifiques à chaque optimiseur, et leur comportement avec différentes fonctions de perte telles que MSE, MAE et Huber, qui sont particulièrement pertinentes pour des tâches de régression. Enfin, nous avons réalisé une recherche d'hyperparamètres sur nos quatre optimiseurs pour identifier lconfigurations optimales de paramètres qui leur sont associées et déterminer quel optimiseur, avec ces paramètres, permet d'obtenir la meilleure MSE.

Cette analyse nous permet non seulement de déterminer quel optimiseur offre les meilleures performances pour ce problème spécifique, mais aussi de comprendre comment les variations dans les paramètres de l'optimiseur affectent l'apprentissage du modèle.

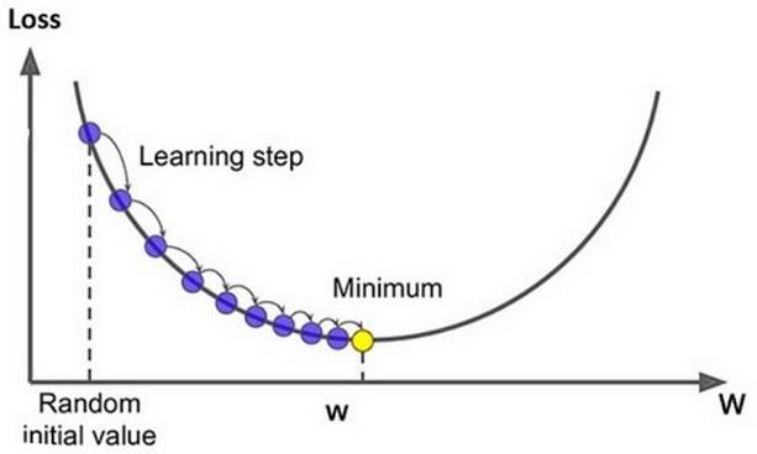
Pour comprendre pleinement ce projet, nous commencerons par définir ce qu'est un optimiseur et expliquer son rôle crucial dans les algorithmes de machine learning. Ensuite, nous présenterons et explorerons les données de la compétition "House Prices- Advanced Regression Techniques". Nous détaillerons les étapes de prétraitement des données que nous avons réalisées, ainsi que le modèles de machine / deep learning que nous avons construits et testés sur plusieurs cas d'optimisation. Enfin, nous évaluerons nos résultats et conclurons sur l'ensemble de notre démarche.

II. Qu'est-ce qu'un optimiseur ?

Un optimiseur est un algorithme ou une méthode utilisée en mathématiques, en informatique et en sciences de l'ingénierie pour ajuster les paramètres d'un modèle afin de minimiser ou maximiser une fonction objective, souvent appelée fonction de coût ou fonction de perte. En apprentissage automatique et en intelligence artificielle, les optimiseurs jouent un rôle crucial dans l'entraînement des modèles, en particulier pour les réseaux de neurones. Leur objectif principal est de trouver les valeurs optimales des poids et des biais du réseau qui minimisent l'erreur de prédiction.

Les optimiseurs fonctionnent en mettant à jour les paramètres du modèle de manière itérative en fonction des gradients de la fonction de perte. Les méthodes les plus courantes incluent la descente de gradient, qui ajuste les paramètres dans la direction opposée au gradient de la fonction de perte par rapport à ces paramètres. Il existe plusieurs variantes de la descente de gradient, telles que la descente de gradient stochastique (SGD), qui utilise des sous-ensembles aléatoires de données pour chaque mise à jour, et des méthodes plus avancées comme Adam (Adaptive Moment Estimation), qui combine les avantages de deux autres extensions de la descente de gradient : AdaGrad et RMSProp.

Schéma :



Le schéma illustre le processus d'optimisation d'un modèle de machine learning avec la descente de gradient et ses variantes. Il montre une courbe de fonction de perte à minimiser, des flèches représentant les gradients calculés, et des points indiquant les valeurs des paramètres à chaque itération. Différentes trajectoires de points démontrent comment les méthodes comme SGD, AdaGrad, RMSProp et Adam ajustent les paramètres pour converger vers le minimum de la fonction de perte.

L'optimisation est essentielle car elle détermine la capacité du modèle à apprendre efficacement et à généraliser à de nouvelles données. Un bon optimiseur permet de converger plus rapidement vers une solution optimale, tout en évitant les pièges courants tels que les minima locaux ou les plateaux où l'apprentissage pourrait stagner. Les choix d'optimiseurs et de leurs hyperparamètres peuvent significativement influencer les performances et la stabilité du modèle final.

Enfin, les optimiseurs sont des outils fondamentaux dans le développement de modèles de machine learning et d'intelligence artificielle. Ils permettent d'ajuster efficacement les paramètres pour atteindre une performance optimale, en minimisant les erreurs et en améliorant la capacité de prédiction des modèles.

III. Présentation des données

Pour la réalisation de ce projet, nous avons décidé utiliser les données de la compétition Kaggle : "House Prices - Advanced Regression Techniques". Centré sur différentes caractéristiques des maisons, ces données visent à être utilisé pour prédire le prix de vente des propriétés : "**SalePrice**".

Description des fichiers :

- **train.csv :** Ensemble de données d'entraînement utilisé pour ajuster les modèles.
- **test.csv :** Ensemble de données de test utilisé pour évaluer les performances des modèles.
- **data_description.txt :** Description complète de chaque colonne, initialement préparée par Dean De Cock et légèrement éditée pour correspondre aux noms de colonnes utilisés ici.
- **sample_submission.csv :** Exemple de soumission basé sur une régression linéaire utilisant l'année et le mois de vente, la superficie du lot, et le nombre de chambres à coucher.

Champs de données :

Voici un aperçu des principaux champs de données présents dans le jeu de données :

SalePrice : Prix de vente de la propriété en dollars (variable cible).
MSSubClass : Classe du bâtiment.
MSZoning : Classification générale de la zone.
LotFrontage : Pieds linéaires de rue connectés à la propriété.
LotArea : Superficie du lot en pieds carrés.
LotShape : Forme générale de la propriété.
Heating : Type de chauffage.
CentralAir : Climatisation centrale.
Kitchen : Nombre de cuisines.
Fireplaces : Nombre de cheminées.
GarageCars : Taille du garage en capacité de voitures.
GarageArea : Surface du garage en pieds carrés.
PoolArea : Surface de la piscine en pieds carrés.
YrSold : Année de vente.
SaleType : Type de vente.
Etc…

IV. Exploration des données

L'exploration des données est une étape cruciale dans tout projet faisant appel à la data, car elle permet de comprendre la structure des données et de déceler les anomalies pour aboutir de manière efficace à l'objectif final. Dans cette section, nous avons analysé les données en nous concentrant sur l'exploration des ensembles de données d'entraînement et de test, ainsi que sur les caractéristiques importantes telles que les corrélations et les valeurs manquantes.

Exploration de l'Ensemble d'Entraînement :

L'ensemble d'entraînement contient **1460 entrées** réparties sur **80 colonnes**. Voici les principales informations :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 80 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   MSSubClass     1460 non-null   int64
 1   MSZoning       1460 non-null   object
 2   LotFrontage    1201 non-null   float64
 3   LotArea        1460 non-null   int64
 4   Street         1460 non-null   object
 ... ... ... ...
 79  SalePrice      1460 non-null   int64
dtypes: float64(3), int64(34), object(43)
memory usage: 912.6+ KB
```

Principalement, les données sont de type **int64**, **float64** et **object**.

Exploration de l'Ensemble de Test :

L'ensemble de test contient **1459 entrées** et **267 colonnes**. Il contient également des informations détaillées sur les propriétés qui sont similaires à ceux du train, mais il n'inclut pas la variable cible **SalePrice**.

Corrélation des Colonnes avec SalePrice :

L'analyse de corrélation permet d'identifier les variables les plus influentes sur la variable cible **SalePrice**. Voici quelques corrélations importantes :

YearRemodAdd	0.507101
YearBuilt	0.522897
TotRmsAbvGrd	0.533723
FullBath	0.560664
1stFlrSF	0.605852
TotalBsmtSF	0.613581
GarageArea	0.623431
GarageCars	0.640409
GrLivArea	0.708624
OverallQual	0.790982
SalePrice	1.000000

Ces valeurs montrent que des caractéristiques comme la surface habitable au-dessus du sol (**GrLivArea**), le nombre de voitures pouvant être garées dans le garage (**GarageCars**), et la qualité globale des matériaux et des finitions (**OverallQual**) ont une forte corrélation avec le prix de vente.

1. Pourcentage de Valeurs Manquantes

Certaines colonnes du jeu de données contiennent des valeurs manquantes, ce qui peut affecter la qualité des modèles prédictifs. Voici les pourcentages de valeurs manquantes pour quelques colonnes clés :

Electrical	0.068493
MasVnrArea	0.547945
BsmtCond	2.534247
BsmtFinType1	2.534247
BsmtQual	2.534247
BsmtExposure	2.602740
BsmtFinType2	2.602740
GarageType	5.547945
GarageQual	5.547945
GarageFinish	5.547945
GarageCond	5.547945
GarageYrBlt	5.547945
LotFrontage	17.739726
FireplaceQu	47.260274
MasVnrType	59.726027
Fence	80.753425
Alley	93.767123
MiscFeature	96.301370
PoolQC	99.520548

Sur les 80 colonnes du jeu de données d'entraînement, 19 présentent des valeurs manquantes. Les colonnes avec un pourcentage élevé de valeurs manquantes **(>10%)** sont : **LotFrontage**, **FirePlaceQu**, **MasVnrType**, **Fence**, **Alley**, **MiscFeature** et **PoolQC**. De plus même si le reste de colonne possédé peux de valeur manquante, l'ensemble de ces colonnes identifier nécessiteront une attention particulière lors du prétraitement des données.

L'exploration des données montre que les ensembles de données d'entraînement et de test contiennent de nombreuses variables pertinentes pour la prédiction des prix de vente des propriétés. Cependant, il y a aussi des défis à relever, notamment la gestion des valeurs manquantes et l'identification des variables les plus influentes.

V. Preprocessing

Le preprocessing des données est une étape cruciale pour préparer les données avant de les utiliser dans un modèle de machine learning. Voici les différentes actions réalisées pour cette étape :

Remplissage des Valeurs Manquantes :

Pour les colonnes avec des valeurs manquantes, nous avons utilisé des valeurs logiques pour les remplir. Par exemple, pour les colonnes catégorielles, nous avons introduit la valeur 'no' pour indiquer l'absence de caractéristique correspondante. Pour les colonnes numériques, nous avons opté pour la valeur 0, signifiant soit l'absence de caractéristique, soit une valeur nulle. Cette approche nous a permis de compléter toutes les valeurs manquantes pour l'ensemble de nos colonnes identifiées, à l'exception de **LotFrontage** et **d'Electrical**.

LotFrontage représente la longueur linéaire en pieds de la rue connectée à la propriété, c'est-à-dire la mesure de la largeur du terrain en contact direct avec la rue. Quant à **Electrical**, il indique le type de système électrique installé dans la propriété, avec des variations en termes de sécurité, d'efficacité et de capacité. Il n'est donc pas possible de remplir ces colonnes en utilisant la même logique que pour les autres.

Suppression de Colonnes :

Nous avons décidé de supprimer les colonnes **LotFrontage** et **Electrical** pour plusieurs raisons. Pour **LotFrontage**, cette colonne présente un taux de valeurs manquantes de **17,74 %**, ce qui est relativement élevé. En d'autres termes, environ un cinquième des observations manquent cette information. La suppression de cette colonne se justifie par la difficulté de remplir ces valeurs avec précision sans introduire de biais. Concernant la colonne **Electrical**, bien qu'elle ne comporte qu'une seule valeur manquante (soit **0,068 %**), son impact potentiel sur la prédiction du prix de vente est probablement limité. Les variations dans les systèmes électriques peuvent ne pas avoir une influence significative sur le prix global d'une maison comparativement à d'autres variables telles que la surface habitable ou la qualité globale de la construction.

Conversion de Type de Colonne :

Nous avons converti la colonne **MSSubClass** de type **Int** à **String** car elle ne représente pas des quantités mais des catégories. Par exemple, les valeurs peuvent indiquer différents types de bâtiments, comme 20 pour une maison unifamiliale d'un étage construite après 1946, 30 pour une maison unifamiliale d'un étage construite avant 1945, etc. En traitant les données catégorielles dans un modèle de machine learning, il est souvent nécessaire de les encoder correctement.

One-Hot Encoding :

Nous avons effectué un one-hot encoding des variables catégorielles pour les jeux de données d'entraînement (**df_train**) et de test (**df_test**), en supprimant la première catégorie pour éviter la multicolinéarité. Ensuite, notre code aligne les colonnes des deux ensembles de données pour s'assurer qu'ils ont les mêmes colonnes, en ajoutant des zéros pour les colonnes manquantes dans l'ensemble de test. Enfin, il vérifie si la colonne **SalePrice** est présente dans l'ensemble de test et la supprime le cas échéant.

Séparation des Variables de Caractéristiques et de la Variable Cible :

Nous avons séparé les données d'entraînement (**df_train**) en variables de caractéristiques (**X**) et variable cible (**y**) en supprimant la colonne **SalePrice** de **X** et en l'assignant à **y**. Ensuite, nous avons aligné les colonnes des ensembles de données d'entraînement et de test (**df_train** et **df_test**) pour nous assurer

qu'ils ont les mêmes colonnes, en ajoutant des zéros pour les colonnes manquantes dans l'ensemble de test.

VI. Construction d'un réseau de neurones

Les principaux travaux réalisés sur ce dataset pour cette compétition utilisent principalement des modèles de régression linéaire classique en raison des consignes de participation. Cependant, dans notre cas, nous avons décidé de ne pas suivre ces règles car l'objectif principal de notre participation à cette compétition était, d'une part, de pouvoir collecter facilement des données et, d'autre part, de pouvoir réaliser une démarche peu voire quasiment pas utilisée dans cette compétition, à savoir la conception d'un modèle de réseau de neurones pour les problèmes de régression, entraîné avec plusieurs optimiseurs et testé à travers plusieurs séries de tests en modifiant certains paramètres impactant l'efficacité de ces optimiseurs.

1. Fixation de la graine aléatoire pour la reproductibilité :

Pour assurer la reproductibilité des résultats, nous avons fait le choix de fixer une graine aléatoire. Cela permet de s'assurer que les mêmes résultats sont obtenus à chaque exécution du code, ce qui est crucial pour s'assurer que notre model soit identique à chaque test réalisé.

2. Normalisation des données :

En utilisant **StandardScaler**, nous avons mis à l'échelle les données d'entraînement, de validation et de test. Cette transformation permet d'harmoniser les différentes échelles des variables, facilitant ainsi l'apprentissage du modèle.

3. Transformation logarithmique de la variable cible :

Pour mieux modéliser la variable cible **SalePrice** et réduire la variance, nous avons appliqué une transformation logarithmique à cette variable. Cette approche est souvent utilisée lorsque la distribution des prix est asymétrique ou présente des valeurs extrêmes.

4. Création du modèle de réseau de neurones :

Nous avons conçu un modèle de réseau de neurones en utilisant l'API **Keras**. Le modèle commence par une couche d'entrée qui prend les données prétraitées et normalisées en entrée. La première couche **dense** (fully connected) contient **256** neurones avec une fonction d'activation **ReLU** (Rectified Linear Unit). Cette fonction d'activation est couramment utilisée dans les réseaux de neurones profonds en raison de sa capacité à introduire de la non-linéarité tout en évitant le problème du gradient qui disparaît. La couche **dense** est suivie d'une couche de normalisation par lots (**BatchNormalization**) qui stabilise et accélère l'apprentissage en normalisant les activations de chaque lot.

Ensuite, une couche de **dropout** avec un taux de **30 %** est ajoutée pour prévenir le surapprentissage en désactivant aléatoirement un pourcentage des neurones à chaque itération. Cette séquence couche dense, normalisation par lots, et dropout est répétée avec des tailles de couche décroissantes : **128**, **64**, et enfin **32** neurones.

Chaque couche **dense** utilise un régularisateur **L2** pour pénaliser les poids de grande amplitude, ce qui aide également à prévenir le surapprentissage. Les initialisateurs **GlorotUniform** sont utilisés pour les poids, garantissant une distribution optimale des valeurs initiales des poids, ce qui favorise une convergence rapide et stable.

La dernière couche du réseau est une couche **dense** avec **un seul neurone** et une **activation linéaire**. Contrairement aux couches précédentes, cette couche n'utilise pas de fonction d'activation non linéaire

car il s'agit d'un **problème de régression**, et nous souhaitons prédire une valeur continue (le prix de vente).

Le modèle est compilé avec un optimiseur (qui peut être **Adam**, **SGD**, **RMSprop**, ou **Nadam**) et une fonction de perte appropriée (**MSE**). L'optimiseur est chargé de mettre à jour les poids du réseau pour minimiser la fonction de perte, et différents optimisateurs peuvent être testés pour trouver la meilleure performance. Le modèle utilise également la métrique **MSE** (erreur quadratique moyenne) pour évaluer la performance pendant l'entraînement et la validation.

Enfin, l'architecture du réseau de neurones est conçue pour équilibrer la capacité d'apprentissage et la généralisation. En combinant des couches **denses**, des **normalisations** par lots, et des couches de **dropout** avec des techniques de régularisation et des initialisations de poids optimales, ce modèle est bien équipé pour apprendre efficacement des données complexes et prédire les prix de vente des propriétés avec précision.

VII. Optimiseurs

Dans le cadre de la compétition Kaggle "House Prices- Advanced Regression Techniques", il est crucial de sélectionner des optimiseurs appropriés pour maximiser la précision des prédictions des prix de vente des maisons. Le jeu de données comprend une variété de caractéristiques des propriétés qui influencent le prix final, telles que la taille du lot, la qualité des matériaux, l'année de construction, le nombre de chambres, etc. Le but est d'utiliser ces caractéristiques pour construire un modèle prédictif performant. Pour ce faire, nous avons sélectionné quatre optimiseurs : **Adam**, **SGD**, **RMSprop** et **Nadam**. Chacun de ces optimiseurs a des propriétés spécifiques qui les rendent adaptés aux différents aspects de ce problème.

1. Choix des optimiseurs

Pourquoi Adam ?

Adam est un optimiseur qui combine les avantages d'**Adagrad** et de **RMSprop**, en utilisant des moyennes mobiles des gradients et des carrés des gradients. Ce qui rend **Adam** particulièrement utile dans le contexte de cette compétition, c'est sa capacité à s'adapter rapidement aux changements dans les gradients et à converger efficacement même avec des données bruitées et variées. Les caractéristiques des propriétés dans ce dataset sont hétérogènes et comprennent des données continues et catégorielles, certaines étant potentiellement bruitées ou ayant des distributions irrégulières. **Adam**, grâce à son mécanisme d'adaptation du taux d'apprentissage, est capable de gérer cette hétérogénéité de manière robuste, permettant une optimisation plus rapide et stable du réseau de neurones.

Pourquoi SGD ?

SGD est l'un des algorithmes d'optimisation les plus fondamentaux et largement utilisés. Sa simplicité et son efficacité en font un choix de référence pour l'évaluation des performances d'autres optimiseurs. Bien que le **SGD** puisse converger lentement et être sensible aux oscillations, il est très efficace pour explorer de grands espaces de paramètres, ce qui est essentiel pour un modèle avec de nombreuses caractéristiques comme dans ce dataset. En utilisant **SGD**, nous pouvons établir une base de comparaison solide pour évaluer les améliorations apportées par des optimiseurs plus sophistiqués.

Pourquoi RMSprop ?

RMSprop est une amélioration du SGD qui adapte le taux d'apprentissage pour chaque paramètre en utilisant une moyenne mobile des carrés des gradients. Dans le contexte de cette compétition, où les caractéristiques des propriétés peuvent varier considérablement en importance et en influence sur le prix de vente, RMSprop aide à stabiliser les mises à jour des poids, assurant une convergence plus rapide et plus stable. Par exemple, des caractéristiques comme "GrLivArea" (surface habitable au-dessus du sol) peuvent avoir une influence majeure, tandis que d'autres comme "PoolArea" (surface de la piscine) peuvent être moins fréquentes mais toujours importantes. RMSprop est capable de gérer cette variance efficacement, ce qui améliore la performance globale du modèle.

Pourquoi Nadam ?

Nadam est une variante de l'Adam qui intègre la technique de Nesterov pour l'accélération des gradients. En combinant les avantages d'Adam et de Nesterov, Nadam promet une convergence plus rapide et précise. Cette méthode est particulièrement bénéfique pour des réseaux de neurones complexes où une convergence rapide et une réduction du surapprentissage sont cruciales. Dans le cadre de la prédiction des prix des maisons, où il est essentiel de capturer les relations complexes entre les différentes caractéristiques des propriétés, Nadam offre un équilibre optimal entre vitesse de convergence et précision. Sa capacité à ajuster les taux d'apprentissage de manière dynamique et à anticiper les mises à jour des gradients permet d'obtenir des résultats plus précis dans un délai plus court.

2. Paramètres Uniques des Optimiseurs

Chaque Optimiseur possédé ces propres hyperparamètres uniques mais aussi c'est hyperparamètre commun. Il est donc indispensable de comprendre ces hyperparamètres afin d'ajuster notre model :

Adam :

- beta_1: Ce coefficient (valeur par défaut 0.9) contrôle la décroissance exponentielle des moyennes mobiles des gradients. Une valeur proche de 1 signifie que les anciennes valeurs des gradients sont fortement retenues, ce qui permet de lisser les fluctuations des gradients.
- beta_2: Ce coefficient (valeur par défaut 0.999) contrôle la décroissance exponentielle des moyennes mobiles des carrés des gradients. Comme beta_1, une valeur proche de 1 retient davantage les valeurs anciennes, ce qui aide à stabiliser les mises à jour des poids.
- epsilon: Un petit nombre (valeur par défaut 1e-07) ajouté pour éviter la division par zéro lors des calculs. Il assure la stabilité numérique.
- amsgrad: Un booléen (valeur par défaut False) qui, lorsqu'il est activé, utilise l'algorithme AMSGrad, une variante d'Adam qui assure que les moyennes mobiles des carrés des gradients ne diminuent jamais, ce qui peut améliorer la convergence dans certains cas.

SGD :

- momentum: Ce paramètre (valeur par défaut 0.0) accélère le gradient en direction des gradients cumulés précédents, ce qui peut aider à accélérer la convergence et à échapper aux minima locaux.
- nesterov: Un booléen (valeur par défaut False) qui, lorsqu'il est activé, utilise le momentum de Nesterov. Cela permet d'anticiper les mises à jour de gradients, souvent conduisant à une convergence plus rapide et plus stable.

RMSprop :

- **rho:** Facteur de décroissance (valeur par défaut 0.9) pour la moyenne mobile des carrés des gradients. Il contrôle la mémoire des gradients passés, stabilisant ainsi les mises à jour.
- **momentum:** Similaire à **SGD**, ce paramètre (valeur par défaut 0.0) aide à accélérer le processus d'optimisation en prenant en compte les gradients précédents.
- **epsilon:** Comme dans **Adam**, ce petit nombre (valeur par défaut 1e-07) évite la division par zéro.
- **centered:** Un booléen (valeur par défaut False) indiquant si la variance doit être centrée. Centrer la variance peut améliorer la stabilité des mises à jour en réduisant les fluctuations causées par les gradients.

Nadam :

- **beta_1:** Coefficient (valeur par défaut 0.9) pour la décroissance exponentielle des moyennes mobiles des gradients, identique à celui utilisé dans **Adam**.
- **beta_2:** Coefficient (valeur par défaut 0.999) pour la décroissance exponentielle des moyennes mobiles des carrés des gradients, également identique à celui utilisé dans **Adam**.
- **epsilon:** Petit nombre (valeur par défaut 1e-07) pour éviter la division par zéro, comme dans **Adam**. **Nadam** combine les avantages de **Nesterov** et **Adam** pour une convergence plus rapide et précise.

3. Paramètres Communs des Optimiseurs

Les optimiseurs partagent plusieurs hyperparamètres communs qui jouent un rôle crucial dans la manière dont les poids du modèle sont mis à jour pendant l'entraînement.

learning_rate: Le taux d'apprentissage détermine la taille des pas que prend l'optimiseur lors de la mise à jour des poids. Un taux d'apprentissage élevé peut conduire à une convergence rapide mais instable, tandis qu'un taux d'apprentissage bas assure une convergence plus stable mais plus lente.

weight_decay: La régularisation des poids aide à prévenir le surapprentissage en pénalisant les grands poids dans le modèle.

clipnorm: Ce paramètre fixe une norme pour le découpage des gradients, empêchant les mises à jour de poids excessivement grandes qui pourraient déstabiliser l'entraînement.

global_clipnorm: Similaire à clipnorm, mais applique la norme de découpage globalement sur tous les gradients, assurant une mise à jour cohérente des poids.

clipvalue: Fixe une valeur de découpage pour les gradients, limitant l'ampleur des mises à jour des poids pour éviter des changements trop drastiques.

use_ema: L'utilisation de la moyenne mobile exponentielle (**EMA**) permet de lisser les valeurs des poids, ce qui peut améliorer la stabilité du modèle.

ema_momentum: La quantité de momentum utilisée pour l'**EMA**. Une valeur élevée signifie que les valeurs passées des poids sont retenues plus longtemps, aidant à stabiliser l'entraînement.

ema_overwrite_frequency: La fréquence à laquelle les valeurs des poids sont mises à jour avec l'**EMA**, influençant la stabilité et la réactivité du modèle.

loss_scale_factor: Utilisé pour l'échelle des pertes, ce paramètre aide à stabiliser l'entraînement lorsque les pertes sont très petites ou très grandes.

gradient_accumulation_steps: Le nombre d'étapes d'accumulation des gradients avant de mettre à jour les poids. Cela peut être utile pour simuler des tailles de batch plus grandes sans augmenter la mémoire **GPU**.

VIII. Test

Maintenant que notre model a été créée et que nos optimiseurs ont été définit, nous allons réaliser plusieurs séries de test en modifiant des paramètres de notre modèle pour tester les performances de celui-ci sur la MSE.

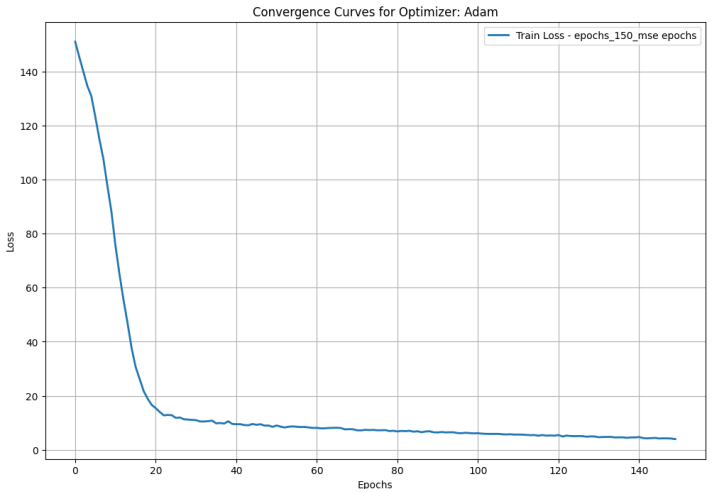
1. Test 1

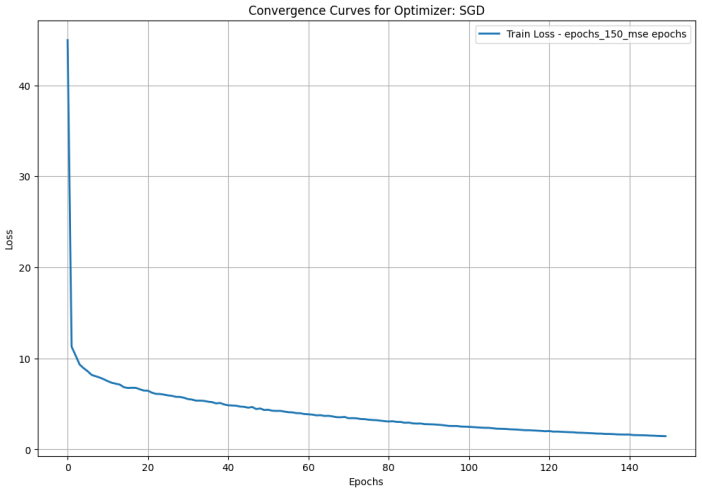
Pour ce premier test, nous avons entraîné notre modèle sur nos quatre optimiseurs sur 150 epochs. Le nombre d'époques est un hyperparamètre très important quant à la bonne performance d'un modèle de machine learning. Voici les résultats obtenus :

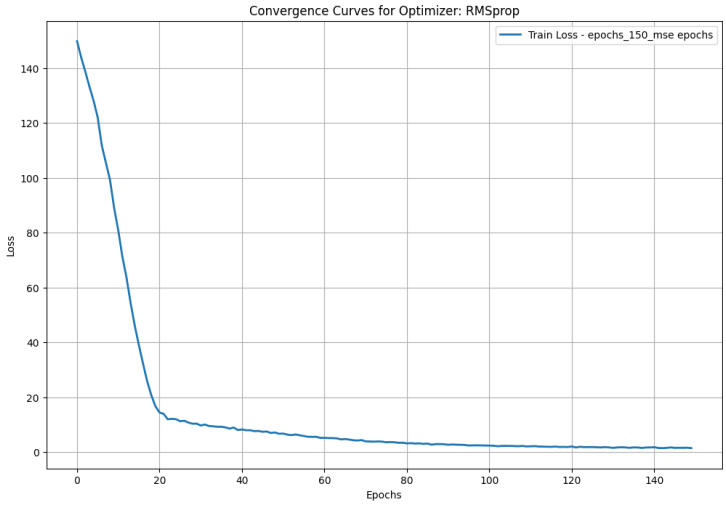
	Optimizer	Configuration	MSE
0	Adam	epochs_150_mse	0.082930
1	SGD	epochs_150_mse	0.040917
2	RMSprop	epochs_150_mse	0.044954
3	Nadam	epochs_150_mse	0.076006

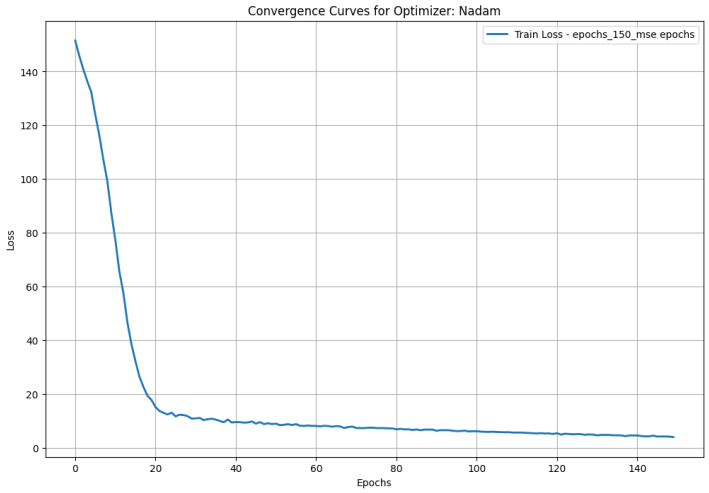
Les résultats de ce test, montre que SGD est premier avec une MSE obtenu de 0.040917.

De plus, si nous observons la convergence de leur loss, nous pouvons voir :









Ici, nous pouvons voir que nos quatre fonctions objectif (loss) avec nos quatre optimiseurs convergent plutôt bien et rapidement. Nous observons que la loss associée à SGD se distingue du lot par une convergence beaucoup plus rapide et linéaire (sans oscillation voir très peux) que les autres. Ce qui soutient et explique pourquoi SGD obtient la meilleure MSE sur ce premier test.

2. Test 2

Pour ce test numéro 2, nous avons testé notre modèle avec nos quatre optimiseurs sur 100 epochs, utilisant trois fonctions de loss différentes (MSE, MAE, Huber) pour chaque optimiseur tout en gardant la même métrique (MSE). Le choix de tester notre modèle avec nos optimiseurs sur ces différentes fonctions de loss est de pouvoir montrer l'impact des résultats du modèle avec nos optimiseurs sur celles-ci. Voici une rapide explication du choix de ces trois fonctions de perte : Le choix de ces trois fonctions de perte (MSE, MAE, Huber) est motivé par notre objectif de maximiser la précision de la prédiction des prix des maisons tout en assurant une robustesse aux anomalies et aux grandes erreurs. En utilisant ces différentes fonctions de perte, nous explorons plusieurs aspects des erreurs de prédiction :

MSE :

La fonction MSE est définie comme la moyenne des carrés des différences entre les valeurs prédites et les valeurs réelles. Mathématiquement, elle est exprimée comme suit :

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

où ***yi*** est la valeur réelle et ***y^i*** est la valeur prédite.

La fonction MSE a été choisie pour son efficacité à pénaliser les grandes erreurs. En minimisant les carrés des différences entre les valeurs prédites et les valeurs réelles, la MSE accorde une importance disproportionnée aux prédictions qui s'écartent fortement des valeurs réelles. Cette propriété est particulièrement utile dans notre contexte car les erreurs importantes peuvent avoir un impact significatif sur l'évaluation globale du modèle. En d'autres termes, la MSE nous aide à obtenir des prédictions plus précises en corrigeant plus fortement les erreurs importantes, ce qui est essentiel pour prédire des prix de vente de maisons de manière fiable.

MAE :

La fonction MAE est définie comme la moyenne des valeurs absolues des différences entre les valeurs prédites et les valeurs réelles. Elle est formulée comme suit :

$$\text{MAE} = \frac{\sum_{i=1}^{n}|y_i - x_i|}{n}$$

La fonction MAE, en revanche, mesure la moyenne des valeurs absolues des erreurs, offrant ainsi une mesure plus intuitive de l'erreur moyenne. Contrairement à la MSE, la MAE ne pénalise pas de manière disproportionnée les grandes erreurs, ce qui la rend moins sensible aux outliers (valeurs aberrantes). Dans notre projet, utiliser la MAE permet de s'assurer que notre modèle est robuste aux anomalies et offre une mesure de performance plus interprétable et réaliste dans des scénarios où quelques données aberrantes peuvent exister. Cela garantit que le modèle ne se concentre pas uniquement sur la réduction des grandes erreurs mais maintient une performance globale équilibrée.

Huber :

La fonction de perte de Huber est définie comme une combinaison des carrés des erreurs pour les petites erreurs et des valeurs absolues des erreurs pour les grandes erreurs. Elle est formulée comme suit :

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,(|y - f(x)| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

La fonction de perte de Huber est une combinaison avantageuse des propriétés de la MSE et de la MAE. Elle est quadratique pour les petites erreurs et linéaire pour les grandes erreurs, offrant ainsi une balance entre pénalisation des grandes erreurs et robustesse aux outliers. Cette propriété est particulièrement utile pour notre projet car elle permet de minimiser les impacts des anomalies tout en maintenant une haute précision pour les prédictions courantes. En utilisant la fonction de perte de Huber, nous pouvons obtenir un modèle qui est à la fois précis et robuste, capable de gérer les variabilités des prix de vente des maisons de manière plus efficace.

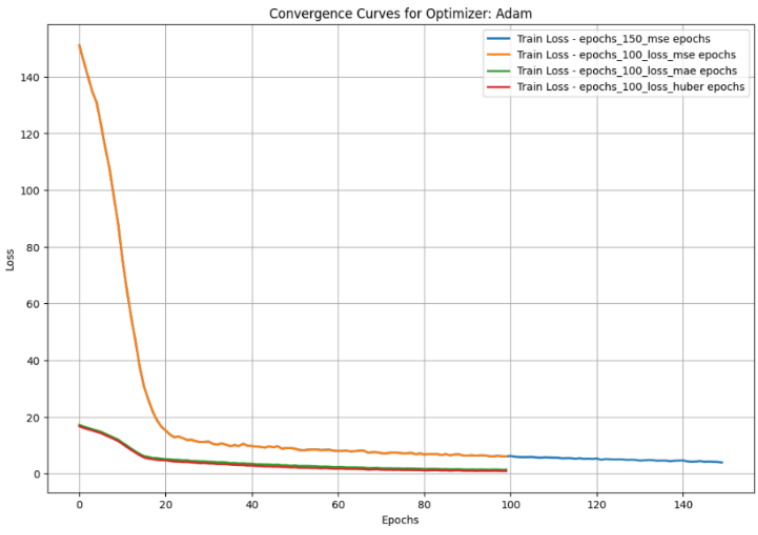
Maintenant, voici les résultats obtenus sur ce deuxième test :

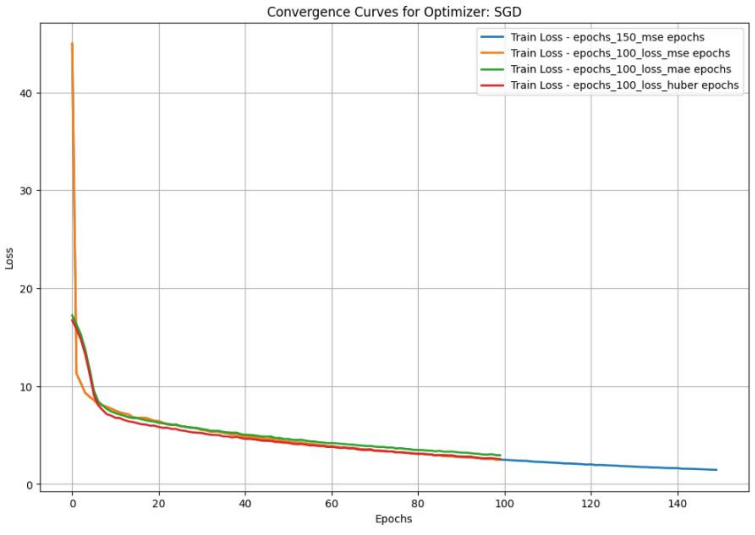
1	Adam	epochs_100_loss_mse	0.106103
2	Adam	epochs_100_loss_mae	0.127578
3	Adam	epochs_100_loss_huber	0.088383
5	SGD	epochs_100_loss_mse	0.064190
6	SGD	epochs_100_loss_mae	0.094046
7	SGD	epochs_100_loss_huber	0.110778
9	RMSprop	epochs_100_loss_mse	0.085324
10	RMSprop	epochs_100_loss_mae	0.103010
11	RMSprop	epochs_100_loss_huber	0.074884
13	Nadam	epochs_100_loss_mse	0.095819
14	Nadam	epochs_100_loss_mae	0.174120
15	Nadam	epochs_100_loss_huber	0.110028

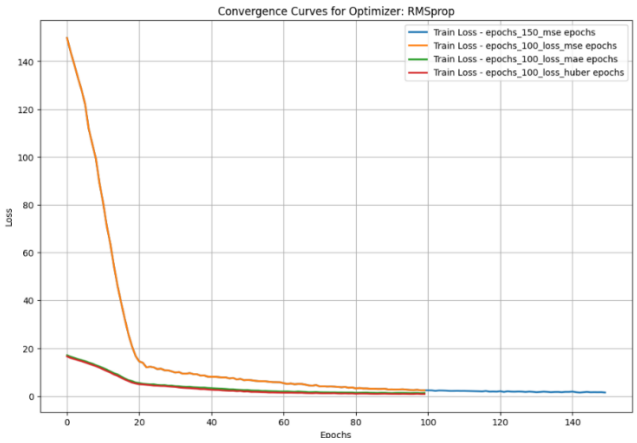
Comparaison des résultats :

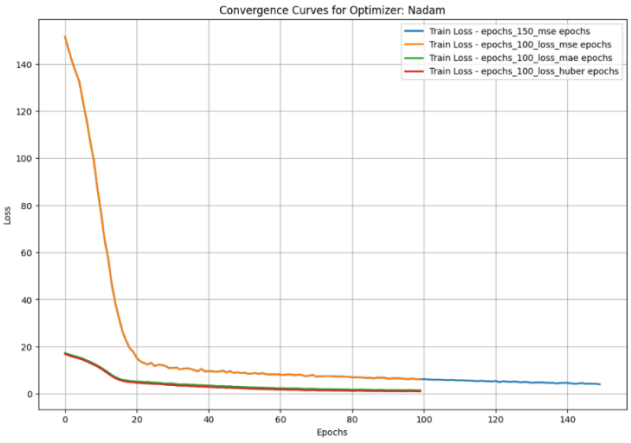
D’après les résultats obtenus, nous pouvons voir qu'Adam obtient de meilleurs résultats lorsqu'il est combiné avec Huber comme fonction de loss, avec une MSE de 0.088383. Pour SGD, nous observons que les meilleurs résultats sont obtenus avec la fonction de loss MSE : 0.064190 de MSE. Pour RMSprop, nous observons des résultats relativement proches entre les fonctions de loss MSE et Huber, bien que Huber produise de meilleurs résultats : 0.074884 de MSE. Enfin, pour Nadam est le meilleur résultat est obtient avec la fonction de loss MSE : 0.095819 de MSE. Maintenant, si nous comparons les meilleurs résultats pour chacun de ces optimiseurs, nous voyons que SGD reste en tête du classement avec une MSE de 0.064190, lorsqu’il est associé à la fonction de perte MSE. Cela suggère que SGD est particulièrement efficace pour cette tâche, peut-être en raison de sa capacité à converger de manière stable sans les ajustements complexes de taux d'apprentissage qui caractérisent Adam et ses variantes.

Si nous comparons les différentes courbes des fonctions de perte utilisé sur chacun de ces optimiseurs, voici les résultats obtenus :









Pour l'optimiseur **Adam**, la fonction de perte **MSE** (jaune) chute rapidement au début de l'entraînement avant de se stabiliser, indiquant une diminution rapide des erreurs suivie d'une convergence constante. Les fonctions de perte **MAE** (vert) et **Huber** (rouge) affichent une baisse moins abrupte, mais une réduction régulière sur la durée, ce qui suggère une convergence plus douce et peut-être plus stable.

Avec **SGD**, les fonctions de perte (**MSE**, **MAE**, **Huber**) révèlent un patron similaire : une réduction rapide de la perte au début, suivie par une descente plus modérée vers la convergence. Les courbes tendant vers des valeurs similaires indiquent une moindre sensibilité du modèle aux fonctions de perte utilisée.

L'optimiseur **RMSprop** montre avec **MSE** (jaune) une forte perte initiale qui diminue rapidement, similaire à **Adam**, suggérant une réactivité efficace aux grandes erreurs initiales. **MAE** (vert) et **Huber** (rouge) démontrent une convergence initialement plus lente mais qui se stabilise à un niveau bas, ce qui peut favoriser une meilleure généralisation et éviter le surajustement.

Pour **Nadam**, la perte initiale avec **MSE** (jaune) est élevée et décroît rapidement, bien que la convergence semble plus lente que celle observée avec **RMSprop** ou **SGD**. Les fonctions **MAE** (vert) et **Huber** (rouge) présentent des courbes similaires entre elles, indiquant un traitement uniforme de ces pertes par **Nadam**.

3. Test 3

Pour ce troisième test, nous avons entraîné à nouveau notre modèle en utilisant nos optimiseurs tout en modifiant le taux d'apprentissage. Les taux d'apprentissage utilisés pour ce test ont été les suivants : un taux **très petit** de **0.001** pour une convergence lente, un autre **modéré** de **0.01** pour équilibrer la vitesse de convergence et la précision de l'approche du minimum, et enfin un taux **élevé** de **0.1** pour une convergence très rapide.

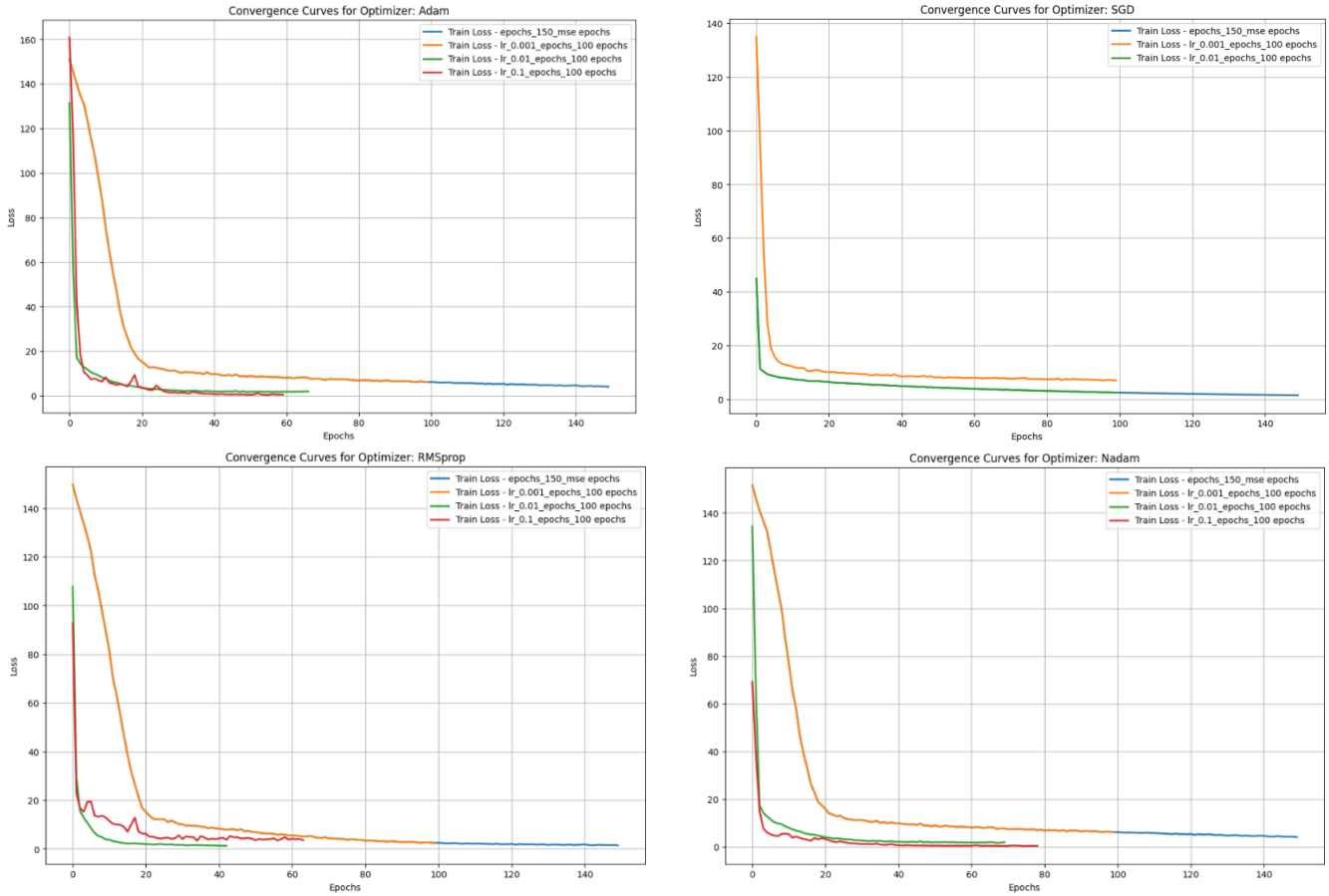
Voici les résultats obtenus pour ce test :

4	Adam	lr_0.001_epochs_100	0.106103
5	Adam	lr_0.01_epochs_100	0.058173
6	Adam	lr_0.1_epochs_100	0.078222
11	SGD	lr_0.001_epochs_100	0.107626
12	SGD	lr_0.01_epochs_100	0.064190
17	RMSprop	lr_0.001_epochs_100	0.085324
18	RMSprop	lr_0.01_epochs_100	0.064714
19	RMSprop	lr_0.1_epochs_100	0.181680
24	Nadam	lr_0.001_epochs_100	0.095819
25	Nadam	lr_0.01_epochs_100	0.059229
26	Nadam	lr_0.1_epochs_100	0.061424

Dans ces résultats, nous constatons d'abord que **SGD** ne fonctionne pas bien avec un learning rate de **0.1** d'où l'absence de résultat. En effet, un learning rate trop élever peut induire des oscillations importantes autour du minimum, rendant difficile la convergence du modèle. Dans certains cas, **SGD**

peut même présenter des problèmes de divergence, où l'erreur augmente plutôt que de diminuer. Cette sensibilité élevée au learning rate constitue un inconvénient majeur de **SGD**. Néanmoins, nous observons que pour chaque combinaison modèle-optimiseur, les meilleurs résultats sont obtenus avec un learning rate modéré de **0.01**. Les meilleurs résultats obtenus sont les suivants : une **MSE** de **0.058173** pour **Adam**, **0.064190** pour **SGD**, **0.064714** pour **RMSprop**, et **0.059229** pour **Nadam**. Le meilleur résultat de ce test est attribué à **Nadam**, avec une **MSE** de **0.059229**, obtenu avec un learning rate de **0.01**.

Maintenant, intéressons-nous au courbe des fonction de perte pour ce test :



Adam :

- **LR = 0.1 (rouge):** Le taux d'apprentissage élevé entraîne une descente très rapide de la perte, mais elle devient instable et oscille fortement à un niveau bas, ce qui peut indiquer que le taux est trop élevé, provoquant des sauts excessifs dans l'espace des paramètres.
- **LR = 0.01 (vert):** Ce taux d'apprentissage montre une diminution rapide et relativement stable de la perte, indiquant un bon équilibre entre la vitesse de convergence et la stabilité.
- **LR = 0.001 (orange):** Ce taux plus faible montre une décroissance plus lente mais très stable de la perte, ce qui peut être préférable pour éviter le surajustement et assurer une convergence fiable à long terme.

SGD :

- **LR = 0.01 (vert):** La perte décroît de manière rapide et reste à un niveau bas de manière plus stable.
- **LR = 0.001 (orange):** Présente la décroissance la plus lente qui converge de manière stable malgré les quelque oscillation identifié.

RMSprop :

- LR = 0.1 (rouge): La perte chute rapidement, mais devient très instable, indiquant des oscillations potentiellement nuisibles pour la performance du modèle final.
- LR = 0.01 (vert): Offre une bonne décroissance est convergence.
- LR = 0.001 (orange): Ce taux le plus faible montre une réduction progressive et stable, sans les fluctuations observées avec les taux plus élevés.

Nadam :

- LR = 0.1 (rouge): Similaire à RMSprop, ce taux provoque une baisse rapide suivie d'une grande instabilité, potentiellement préjudiciable.
- LR = 0.01 (vert): La décroissance est rapide et reste relativement stable, offrant une convergence adéquate.
- LR = 0.001 (orange): Présente la courbe la plus lente est régulière, favorisant une convergence sans risque de sauts ou d'instabilité.

4. Test 4

Pour ce quatrième test, nous avons entrainer une nouvelle fois notre model sur 100 epochs avec nos mêmes optimiseurs. Cependant, nous avons modifié les hyperparamètre unique de chacun de nos optimiseurs pour voir les différentes performances :

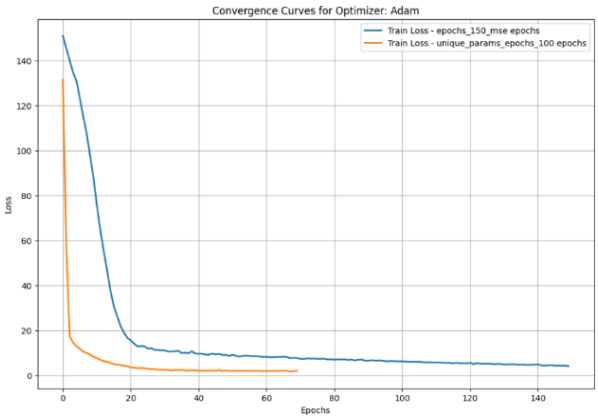
- Adam: learning_rate: 0.01, beta_1: 0.9, beta_2: 0.999, epsilon: 1e-07, amsgrad: False
- SGD: learning_rate: 0.01, momentum: 0.9, nesterov: True
- RMSprop: learning_rate: 0.01, rho": 0.9, momentum: 0.9, epsilon: 1e-07, centered: True
- Nadam: learning_rate: 0.01, beta_1: 0.9, beta_2: 0.999, epsilon: 1e-07

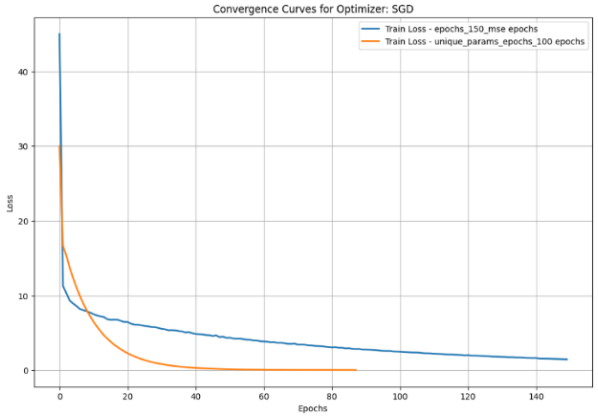
Voici les résultats obtenus :

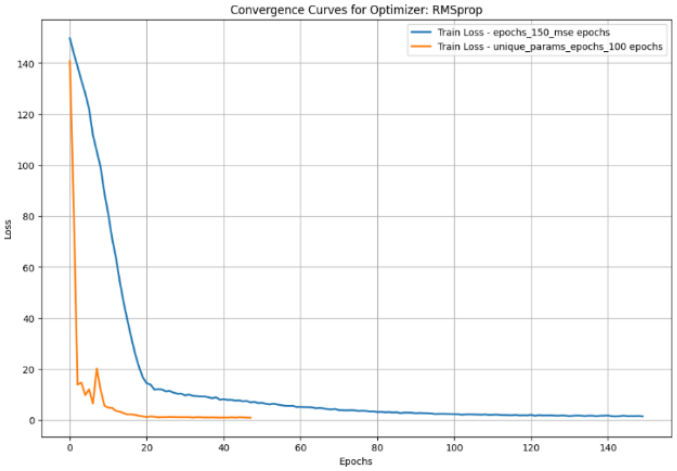
```
1       Adam  unique_params_epochs_100  0.062896
3        SGD  unique_params_epochs_100  0.058760
5    RMSprop  unique_params_epochs_100  0.062496
7     Nadam  unique_params_epochs_100  0.071613
```

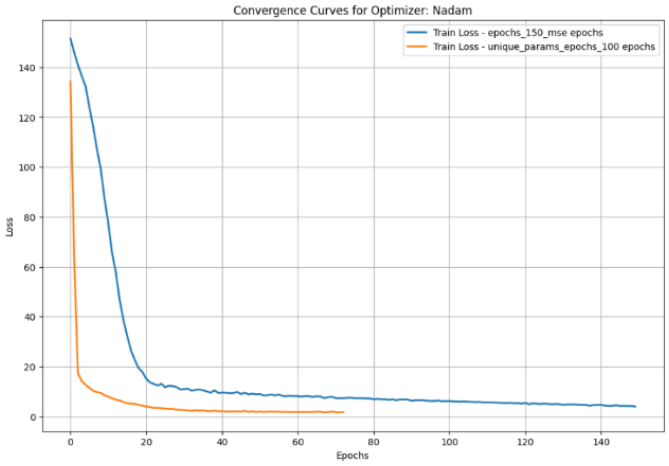
En modifiant nos paramètres, nous pouvons déjà constater une nette amélioration comparée aux résultats obtenus sur 150 epochs avec les optimiseurs suivants : pour Adam avec une MSE de 0.062896 contre 0.082930 sur 150 epochs ; pour Nadam avec une MSE de 0.071613 contre 0.076006 sur 150 epochs. Cela nous montre que chaque paramètre de nos optimiseurs joue un rôle crucial quant à la performance de notre modèle. Dans le cas de ce test, le meilleur résultat se concentre sur SGD, avec une MSE de 0.058760.

Maintenant intéressons-nous au graphe de nos fonction de perte :









Adam : Le graphique montre que le modèle converge rapidement, avec une chute significative de la perte durant les premières époques. Cela indique une adaptation efficace des poids du modèle aux données. La courbe s'aplatit après environ 30 époques, suggérant que les améliorations supplémentaires sont marginales. Ceci est cohérent avec la capacité d'**Adam** à gérer efficacement les gradients dispersés grâce à ses estimations adaptatives des moments de premier et second ordre.

SGD: Le modèle utilisant **SGD** montre une courbe similaire avec une baisse rapide de la perte initiale et une convergence stable par la suite. L'utilisation du momentum et du nesterov dans **SGD** semble améliorer la capacité de l'optimiseur à éviter les minima locaux comme en témoigne la pente initiale.

RMSprop: Ici, la courbe est similaire à celle d'**Adam**, mais avec quelques instabilités entre 0 et 10 épochs, indiquant des ajustements fréquents dans les mises à jour des poids qui pourraient refléter une réponse à des caractéristiques problématiques des données. **RMSprop** est souvent préféré pour des applications où les gradients peuvent varier considérablement, ce qui pourrait expliquer ces fluctuations.

Nadam: Nadam montre une courbe de convergence similaire à celle d'**Adam** mais avec une légère instabilité vers la fin. Cette instabilité pourrait indiquer que le modèle commence à réagir excessivement aux particularités des données, peut-être en surajustement.

5. Test 5

Pour ce cinquième et dernier test, nous avons utilisé **Keras Tuner** pour rechercher les hyperparamètres optimaux de nos optimiseurs afin d'améliorer de manière drastique les performances de notre modèle. À cette fin, nous avons testé **100 combinaisons d'hyperparamètres**, avec un test réalisé par configuration. Cette recherche a été effectuée sur **150 epochs**, en utilisant le même modèle que précédemment.

Voici les résultats obtenus :

```
Trial 100 Complete [00h 00m 44s]
val_mse: 0.0448736809194088

Best val_mse So Far: 0.029791157692670822
Total elapsed time: 01h 16m 21s
```

```
The hyperparameter search is complete. The optimal hyperparameters are:
Optimizer: nadam
Learning Rate: 0.008850221174397929
Beta 1: 0.8300000000000001
Beta 2: 0.9390000000000001
Epsilon: 1.1037353035334539e-07
```

L'optimiseur **Nadam** a été sélectionné comme le plus efficace, avec un taux d'apprentissage de 0.008850, qui est inférieur au taux courant (modéré : 0.01) pour un apprentissage détaillé tout en

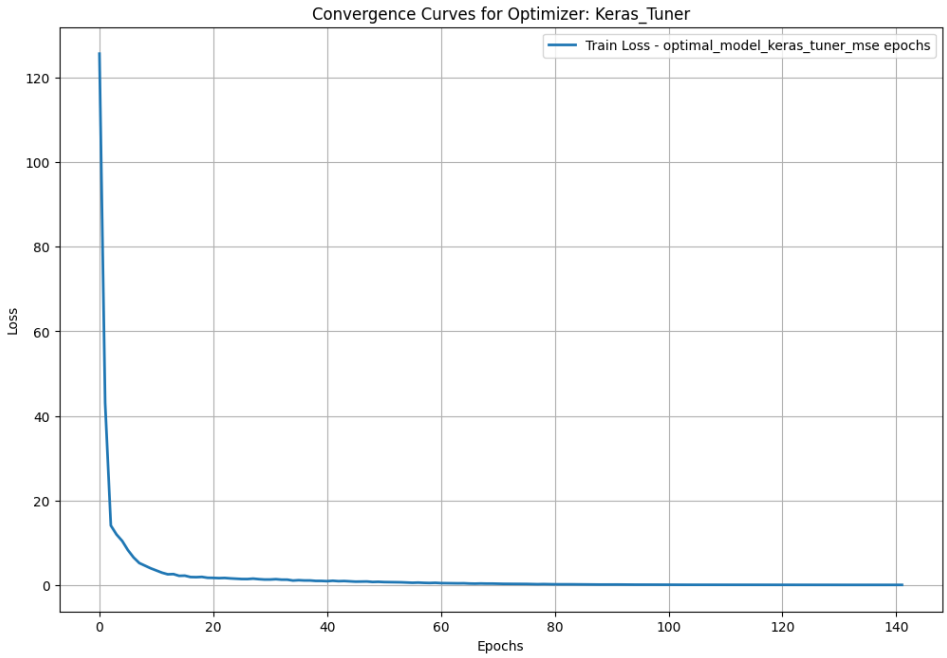
maintenant une progression rapide. Les paramètres Beta 1 et Beta 2 ont été ajustés à 0.83 et 0.939 respectivement, ce qui permet une réactivité modérée aux gradients les plus récents tout en préservant une accumulation efficace du carré des gradients, contribuant à des mises à jour de poids plus lisses et précises. L'Epsilon très bas, environ 1.10e-07, assure la stabilité numérique lors de ces mises à jour. Ces ajustements ont mené à une amélioration de l'erreur quadratique moyenne sur les données de validation, la réduisant à **0.029791**, indiquant une précision accrue du modèle dans la prédiction des prix des maisons.

Maintenant, voici le résultat de l'entrainement du model en utilisant les meilleur hyperparamètre de l'optimiseur trouvé sur **150 epochs**:

```
Keras_Tuner  optimal_model_keras_tuner_mse  0.029001
```

Nous pouvons voir que le résultat obtenu est nettement plus performant que les résultats précédents. Nous passons du meilleur résultat précédent sur le premier test avec **SGD**, qui était de **0.040917** de **MSE**, à **Nadam** avec **0.029001** grâce à la recherche d'hyperparamètres. Ce résultat est actuellement le meilleur que nous avons pu obtenir et démontre, comme le quatrième test, l'importance des hyperparamètres des optimiseurs quant aux performances du modèle.

Maintenant, voici le graphique concernant la courbe de perte :



La courbe de convergence pour l'optimisation du modèle révèle une performance prometteuse du modèle optimisé par Keras Tuner avec l'optimiseur Nadam. Cette courbe montre une chute rapide de la perte dès les premières époques, signifiant que le modèle apprend rapidement les caractéristiques pertinentes des données. Après environ 20 époques, la perte se stabilise autour d'une valeur basse, ce qui indique que le modèle a atteint une convergence et qu'il n'y a pas de surapprentissage significatif visible.

IX. Evaluation & Résultats

Les différents tests réalisés précédemment nous permettent de diagnostiquer plusieurs éléments importants concernant la performance de notre modèle. En effet, sur les 5 tests réalisés, nous pouvons voir que notre modèle est plus performant avec **SGD** : pour le nombre d'**épochs** (de **0 à 150**) affichant une **MSE** de **0.040917**, sur la fonction de perte choisie (**MSE**) pour une **MSE** de **0.064190** obtenu, et sur l'implémentation fixe de ces hyperparamètres uniques avec une **MSE** de **0.058760**. Cependant, concernant le taux d'apprentissage, nous avons pu voir que **Nadam** était plus performant dans le cas de cette variation d'hyperparamètres avec une **MSE** de **0.059229** et qu'il a également été le plus efficace lors de la recherche d'hyperparamètres avec **Keras Tuner**, enregistrant la meilleur **MSE** du classement : **0.02979**. De plus, si nous nous intéressons aux résultats concernant les courbes des différentes fonctions

de perte observées, nous pouvons conclure que, en fonction de chaque hyperparamètre modifié, l'impact sur la fonction de perte est significatif. En effet, plusieurs caractéristiques nous le montrent, notamment par le niveau de chute de la courbe, la convergence, l'oscillation, etc. Les résultats obtenus précédemment sont nettement différents de ceux obtenus avec la recherche d'hyperparamètres avec Nadam. Grâce à cette recherche d'hyperparamètres, notre fonction de perte converge beaucoup mieux, nous offrant un niveau de performance nettement plus gratifiant que précédemment. Cela nous prouve, en plus des résultats sur les métriques, que l'optimisation de nos optimiseurs et hyperparamètres a une influence sur la performance de notre fonction de perte et plus généralement sur le modèle quant à son bon fonctionnement et efficacité.

Ces résultats nous permettent de conclure sur plusieurs aspects extrêmement importants : chaque optimiseur est plus ou moins sensible à différents hyperparamètres, d'où l'importance de les tester. En effet, SGD a tendance à être plus sensible au nombre d'**époch**s qu'à la modification de ses hyperparamètres. Adam montre une sensibilité plus accrue à la modification du taux d'apprentissage. Comme pour SGD, RMSprop semble mieux s'optimiser sur le nombre d'**époch**s. Enfin, Nadam est probablement le plus sensible à la modification de ses hyperparamètres dans l'ensemble, dû à ces résultats.

Maintenant, en ce qui concerne la compétition Kaggle, l'évaluation de notre travail est effectuée sur la RMSE grâce au système d'API dont dispose la compétition. La RMSE (Root Mean Squared Error) est une métrique d'erreur utilisée pour mesurer la différence entre les valeurs prédites par un modèle et les valeurs réelles. Elle est calculée en prenant la racine carrée de la MSE, qui est la moyenne des carrés des différences entre les valeurs prédites et réelles. La RMSE est en unités identiques à celles des données observées, ce qui la rend plus interprétable que la MSE.

Maintenant, pour pouvoir soumettre nos résultats, nous avons entraîné notre modèle sur les données de test sur 150 **époch**s avec le meilleur optimiseur trouvé et ses hyperparamètres optimaux, Nadam.

Voici le résultat obtenu avec notre place sur le leaderbord :

Comme l'indique la compétition, nous avons obtenu un score de RMSE de 0,16695, ce qui n'est pas forcément un très bon score, étant donné notre place sur le leaderboard, qui est 3501 sur 4905. Cette position peut s'expliquer en partie par notre objectif qui est de comprendre et de tester différents optimiseurs pour améliorer les performances de notre modèle. De plus, la compétition souligne un problème de régression avancé avec l'emploi de méthodes comme le feature engineering, qui correspond au processus de sélection, de modification, ou de création de nouvelles caractéristiques à partir de données brutes pour améliorer la performance des modèles de machine learning, ce que nous n'avons pas réalisé. Nous pouvons aussi tenter d'augmenter les performances de notre modèle en continuant à réaliser d'autres tests sur d'autres hyperparamètres ou sur les mêmes, en trouvant différents axes d'ajustement. Ou encore, nous pourrions modifier notre réseau de neurones pour en créer un encore plus sophistiqué pour ce problème avec plus de couche et de neurones.

X. Conclusion

Au terme de ce projet, nous avons acquis une compréhension approfondie et réalisé une analyse détaillée du comportement de quatre optimiseurs différents appliqués à un modèle de réseau de neurones. En effectuant de multiples tests pour ajuster les hyperparamètres, nous avons réussi à améliorer les performances de notre modèle dans la prédiction des prix de vente des maisons. Les résultats obtenus sont particulièrement concluants en termes d'analyse, ayant permis d'identifier divers cas de sensibilité relatifs à l'optimisation de nos optimiseurs par le biais de changements diverses des hyperparamètres.

Ainsi ce module d'optimisation convexe a été essentiel pour comprendre l'importance cruciale des optimiseurs dans divers contextes d'ingénierie, comme l'industrie et la data science. Ce fut un ajout très bénéfique à notre formation, car il nous a permis d'explorer et de traiter un aspect important de l'optimisation qui n'avait pas encore été abordé dans nos cours précédents. Enfin, ce projet a donc non seulement enrichi notre compréhension théorique, mais il a également élargi notre perspective pratique sur l'application de techniques avancées dans la machine learning.

XI. Sources

- House Prices- Advanced Regression Techniques | Kaggle
- 10 famous Machine Learning Optimizers- DEV Community
- Comment Choisir la Fonction de Perte- Meilleur Tutoriel (inside-machinelearning.com)
- https://en.wikipedia.org/wiki/Huber_loss
- Loss Functions in Machine Learning Explained | DataCamp
- 7 Most Common Machine Learning Loss Functions | Built In
- Adam: A Method for Stochastic Optimization (arxiv.org)
- Stochastic gradient descent- Wikipedia
- Understanding RMSprop — faster neural network learning | by Vitaly Bushaev | Towards Data Science
- NADAM (serp.ai)
- 134- What are Optimizers in deep learning? (Keras & TensorFlow) (youtube.com)
- Optimization for Deep Learning (Momentum, RMSprop, AdaGrad, Adam)- YouTube
- In Stream Ad- VO V3- Hey ML teams (youtube.com)
- Gradient descent, how neural networks learn | Chapter 2, Deep learning- YouTube
- Gradient Descent. Gradient descent is very generic… | by Ehtisham Raza | Medium