



Année universitaire 2021/2022

Master 1 DAC

**PROJET :**

**Conception d'une librairie sur les réseaux de neurones**

*Machine Learning*

Enseignant : N. Baskiotis

Valentin FONTANGER - 21115619

Hai Nam Le - 28613397

## **Table des matières :**

<b>Introduction</b>	<b>2</b>
<b>Conception de la librairie</b>	<b>2</b>
<b>2. Réseau de neurones</b>	<b>4</b>
<b>3. Classification:</b>	<b>4</b>
<b>4. Auto-encoder:</b>	<b>5</b>
<b>5. Convolution</b>	<b>9</b>
<b>6. Interprétation d'un CNN</b>	<b>17</b>
6.1. MNIST	18
6.2. CIFAR 10	24
6.3. FRUITS-360	32
6.4. IMAGENETTE	38

# Introduction

L'objet de ce projet est de développer une librairie permettant la conception de réseaux de neurones. Connaissant un essor considérable, ce travail s'inscrit dans une démarche de compréhension de cette algorithme puissant trop souvent perçu comme une « boîte noire ».

Afin de comprendre les tenants et les aboutissants des réseaux de neurones, nous avons dirigé plusieurs expérimentations, retranscrites au sein de ce rapport. Ces expérimentations portent sur des architectures de réseaux différentes, des problématiques différentes (compression d'images, classification multi classe pour des problèmes non linéaires, régression, ...), et mettent en avant les améliorations apportées par la recherche au domaine de l'apprentissage profond. Le lecteur trouvera l'intégralité des expériences à l'aide des notebooks mis à dispositions, et chaque exemple et image fera référence à un notebook et la cellule concernée.

Dans un premier temps, nous évoquerons brièvement la méthodologie et la philosophie lors de la conception du projet. Puis, nous évoquerons les différentes expérimentations, de la compression d'image en passant par les réseaux convolutifs. Finalement, nous conclurons sur les atouts et défauts de notre librairie, ainsi que les futures améliorations à apporter aux implémentations et expérimentations.

Voici le dépôt contenant l'intégralité du projet et des expérimentations :

<https://gitlab.com/Nam2371999/pytorchdiy>

## 1. Conception de la librairie

Nous avons souhaité développer un outil aux implémentations robustes. C'est dans cette démarche que nous avons consacré une partie de notre temps aux tests unitaires. Pour ce faire, nous avons comparé nos résultats en **forward** ainsi qu'en **backward**, avec le framework **Pytorch** et son outil de différenciation automatique **autograd**. Ainsi, nous nous sommes assurés que toutes les fonctions d'activations et couches de notre réseau retournent une sortie et un gradient convenable.

```
def test_backward_softmaxce(self):
    x = np.random.randn(self.batch, self.d) # (2, 2)
    _x = torch.tensor(x, requires_grad=True, dtype=torch.float32)

    y_true = np.zeros((self.batch, self.d))
    y_true[:,0] = 1
    _y_true = torch.tensor(y_true, dtype=torch.float32)

    softmaxce = SoftmaxCE()
    oracle_softmaxce = torch.nn.CrossEntropyLoss(reduction="sum")

    d_z = softmaxce.backward(x, y_true)
    output_oracle = oracle_softmaxce(_x, _y_true)
    output_oracle.backward()

    oracle_d_z = _x.grad

    self.assertEqual(d_z.shape, (self.batch, self.d))
    np.testing.assert_almost_equal([d_z, oracle_d_z])
```

### *Test unitaire du calcul du gradient de la SoftmaxCE (nous vs pytorch)*

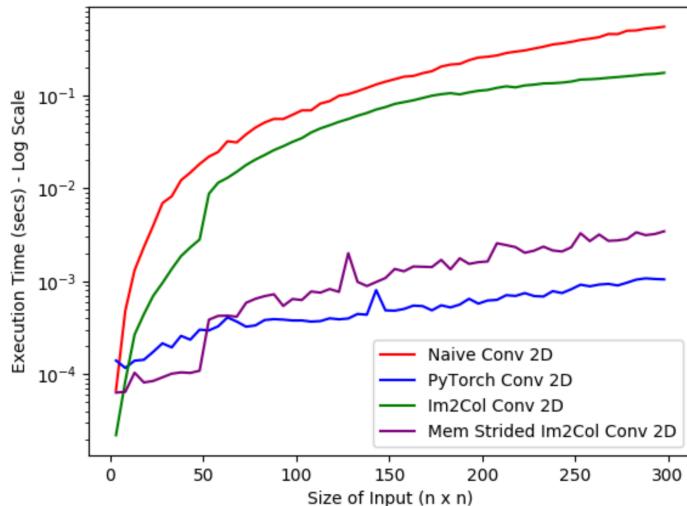
Concernant l'aspect réseaux convolutifs de notre travail, nous avons fait un choix important concernant l'implémentation. En effet, nous avons débuté par une implémentation classique des réseaux convolutifs, en utilisant des boucles for afin de déterminer les sorties et dérivées de nos modules. Cependant, cette approche, bien que fonctionnelle, n'est pas la plus rapide.

Nous avons implémenté la convolution pour les données 1D et 2D. Notre implémentation propose les attributs importants d'une convolution Pytorch, à l'exception de la dilation. Cet effet peut être réalisé en utilisant un large kernel, à un certain coût de calcul. Ce travail fait l'objet d'une amélioration future.

La technique **im2col** permet de stocker dans une matrice, toutes les fenêtres de notre convolution. En aplatisant le kernel, nous avons vectorisé les opérations d'additions et de multiplications de la convolution. Nous avons utilisé la technique **memory stride** par-dessus **im2col** pour un gain de performance. Cette dernière nous évite complètement les boucles lors de l'indexation des coefficients de notre convolution en utilisant la façon dont est stocké un numpy array.

Concernant la passe arrière (**backward pass**), le gradient du kernel est agrégé le long des axes spatiaux du signal rétro propagé, de même taille que la sortie. Le calcul de la dérivée par rapport à l'entrée est représenté par une convolution avec un **stride fractionné**.

Nous pouvons utiliser la même approche pour les opérations de pooling



*Temps d'exécution de la convolution selon différentes approches*

Le gain de temps observé est conséquent, et nous a permis de réaliser plusieurs expérimentations.

Introduisons dès à présent, les différentes expériences menées.

## 2. Réseau de neurones

Nous avons implémenté les couches linéaires, les fonctions d'activations sigmoid, softmax, tanh, relu, les pertes entropy croisées, softmax-cross entropy, MSE et cross entropy binaire. La fonction softmax-CrossEntropy permet d'éviter les erreurs de précisions.

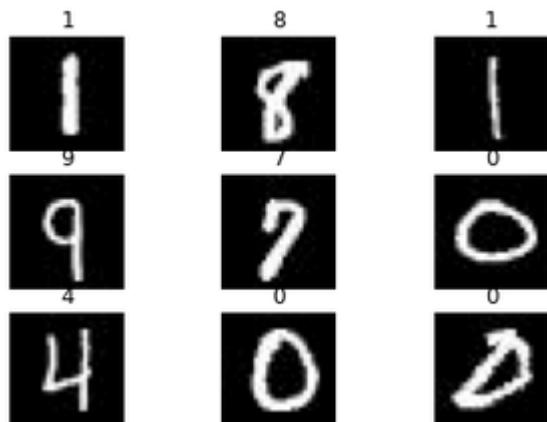
Afin de faciliter l'API, nous avons implémenté une encapsulation pour un réseau de neurones contenant plusieurs couches.

Finalement, nous avons implémenté deux stratégies d'optimisation : la descente de gradient stochastique et Adam. Nous disposons également de plusieurs méthodes d'initialisation.

## 3. Classification:

Nous avons construit un modèle de classification simple pour la tâche de reconnaissance des chiffres pour le jeu de données MNIST. Nous utilisons l'entropie croisée softmax comme fonction de perte, et l'architecture suivante :

Linear(784,128) > ReLU() > Linear(128,10)



Il atteint une précision de 91,9%.

## 4. Auto-encoder:

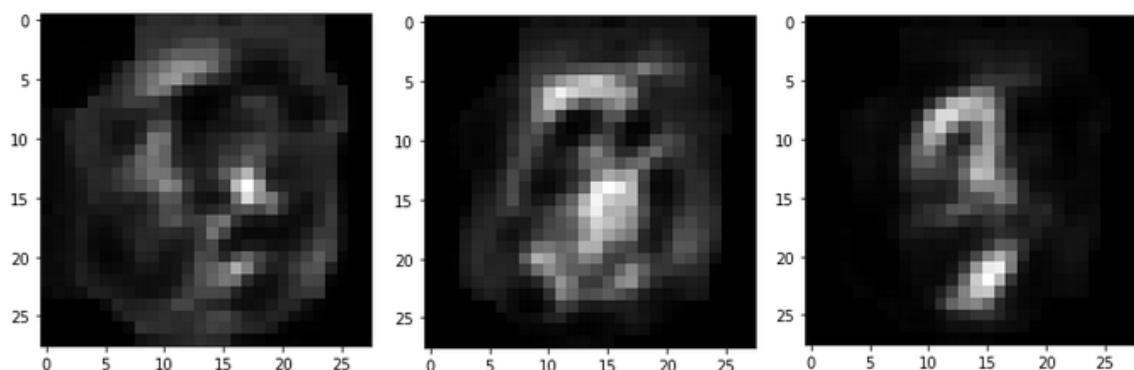


Nous avons conçu un modèle auto-encoder utilisant nos couches linéaires. La tâche est de reconstruire l'image issue du jeu de données MNIST. Avec des images de tailles  $28 * 28 * 1 = 784$ , nous utilisons l'architecture suivante :

Encoder: Linear(784,256) > TanH() > Linear(256,32) > Tanh()

Décoder: Linear(32,256) > TanH() > Linear(256,784) > Sigmoid()

Les résultats de nos expérimentations ont montré leurs sensibilités aux hyperparamètres. Après de nombreux essais erreurs, un learning rate de 0.008, un batch size de 16 et 100 epochs permettent d'obtenir les meilleures résultats. L'évolution de la perte en entraînement nous laisse penser qu'augmenter le nombre d'epoch nous permettrait d'obtenir de meilleures performances



*Activation des neurones de la couche de compression*

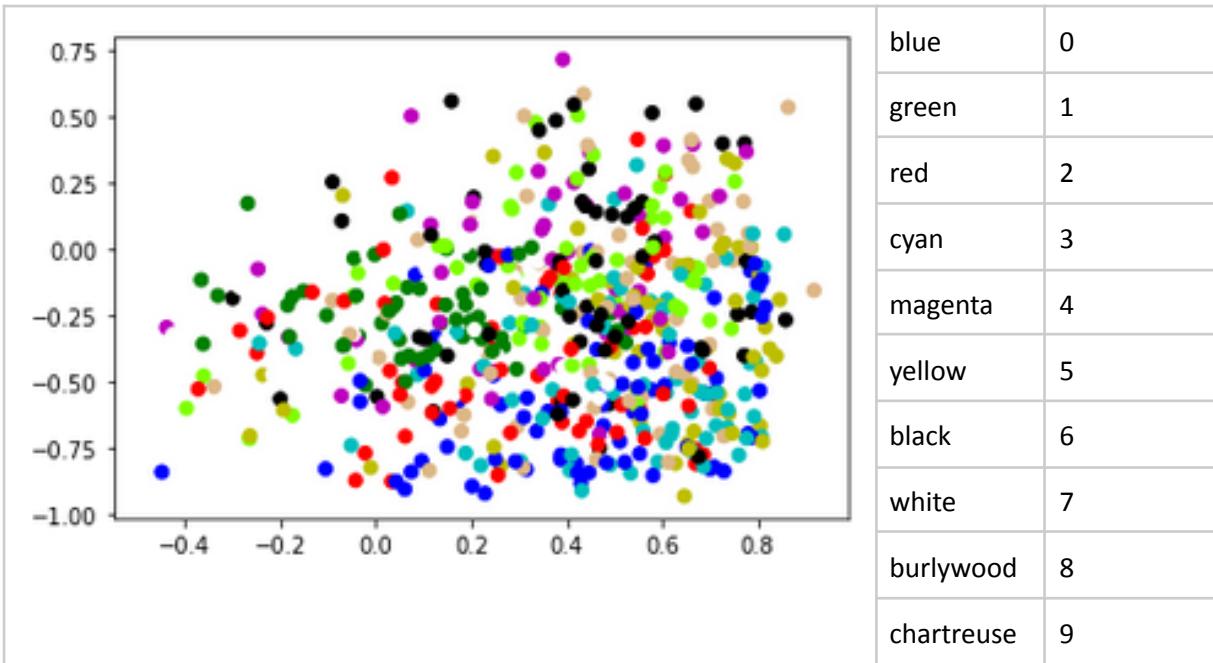
	Prediction	Original
--	------------	----------

Train	0 2	0 3	5 4	0 2	0 3	5 4
	3 4	3 5	4 1	2 4	3 5	4 1
	4 5	5 6	1 2	4 5	5 6	1 2
Test	1 9	8 7	1 0	1 9	8 7	1 0
	9 4	7 0	0 0	9 4	7 0	0 0
	4 5	0 6	3 3	4 5	0 6	3 3

L'auto-encodeur est robuste au bruit gaussien. Lorsqu'il est entraîné avec des images bruitées, l'auto-encodeur parvient à les écarter et à produire des images décodées claires. Cependant, sous le même choix d'hyperparamètres que la version non-bruitée, nous observons visuellement une diminution de la performance.

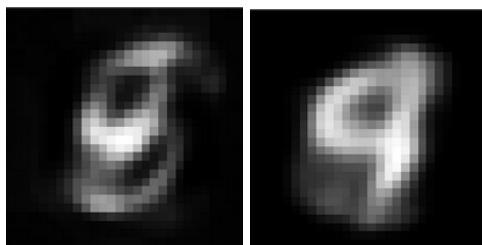
	Prediction			Original		
Train	0 2	0 3	5 4	0 2	0 3	5 4
	3 4	3 5	4 1	2 4	3 5	4 1
	4 5	5 6	1 2	4 5	5 6	1 2

Nous avons essayé de tracer les points de données compressés produits par un auto-encodeur de 2 dimensions latentes.

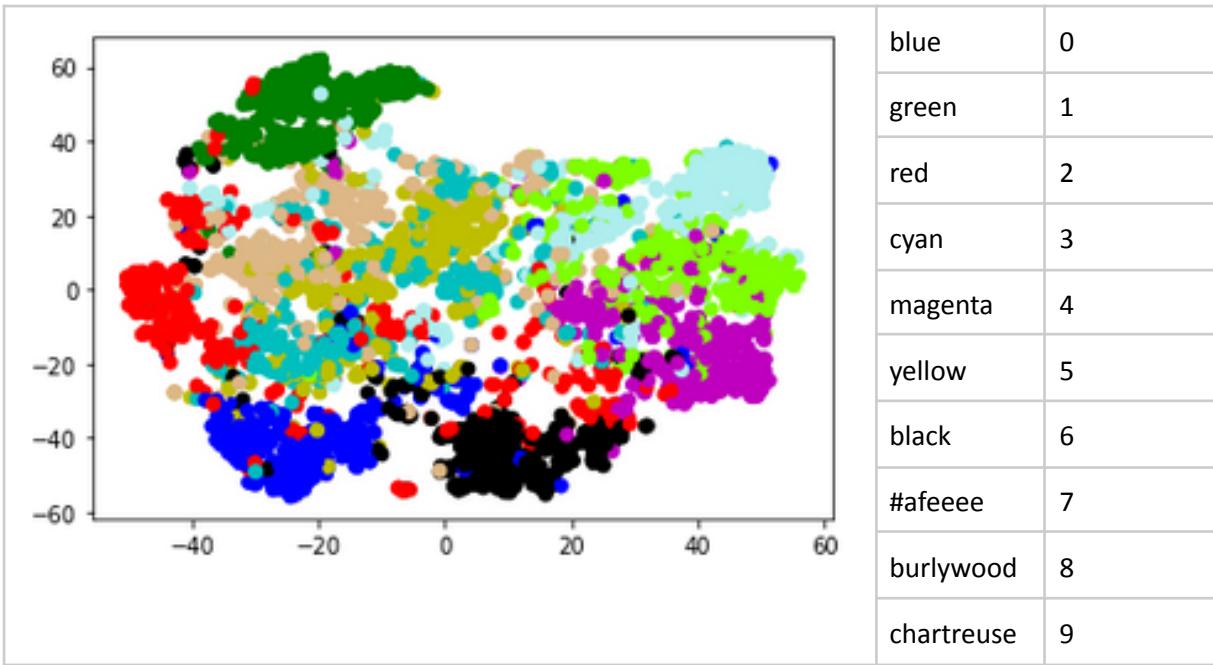


Nous voyons que la classe 0 (bleu) est complètement séparée de la classe 1 (vert), et que la classe 2 (rouge) est quelque part entre les deux. Il n'y a pas de distinctions aussi claires dans les autres classes. Ce résultat est cohérent avec les formes visuelles de ces chiffres.

Voici les activations correspondant aux 2 dimensions latentes



Dans l'espoir d'une meilleure distinction entre les classes, nous utilisons une approche différente. Nous utilisons t-SNE pour projeter les représentations latentes 32-dimensionnelles produites par l'architecture originale.



Nous voyons que les classes sont bien séparées. Cette visualisation suggère que les points de données de la même classe sont regroupés dans l'espace latent à 32 dimensions, alors que les groupes de classes différentes ne s'effondrent pas. Encore une fois, nous voyons que les 0 sont très éloignés des 1, et que les 2 sont quelque part entre les deux.

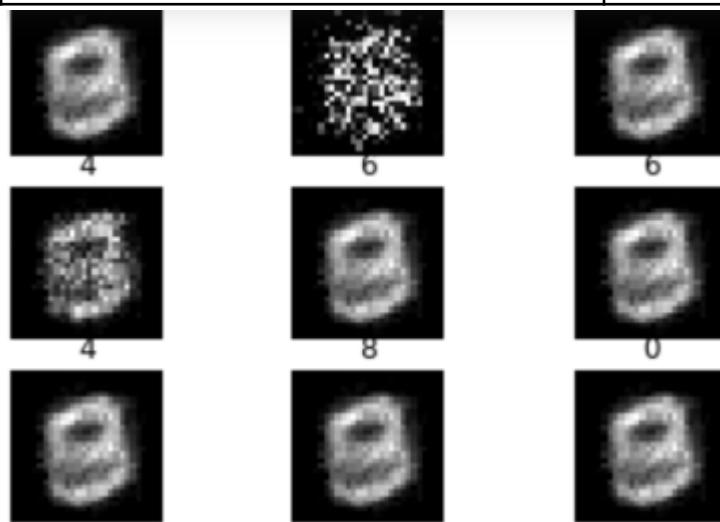
Nous essayons d'utiliser la représentation latente comme features d'entrée pour la tâche de **classification**. Avec 32 dimensions latentes, la même architecture décrite dans la section Classification a obtenu une précision de **83%** pour la formation et de **76%** pour le test. L'intuition pour cette baisse de performance est qu'en compressant les données, nous avons perdu des caractéristiques visuellement discriminantes entre les classes.



Par exemple,  est compressé en , qui, bien que conservant les principales régions blanches de l'image originale, ressemble maintenant beaucoup à un 0.

Durant nos expérimentations, nous avons observé les problèmes suivants :

Piège	Solution
Dépendance entre learning rate et batch size	Agrégé le signal du backward en utilisant la moyenne des échantillons du lot plutôt que la somme
Norme du gradient instable (Disparition ou explosion du gradient)	Une initialisation précautionneuse, nous avons utilisé Xavier pour les couches linéaires
Explosion du gradient due aux valeurs extrêmes de l'entrée passée à la Cross Entropy Binaire.	Nous avons opté pour la méthode du seuillage du gradient (gradient clipping)
Disparition du gradient au niveau des régions saturées de la tanh ou sigmoid.	Seuillage de l'entrée de la sigmoid et de la tanh.



Concernant le problème de la disparition du gradient provoqué par la Sigmoid et la TanH : le gradient appliquée afin de punir les faux pixels blancs diminue à cause de la saturation de ces fonctions. L'effet observé sur les images décodées est intéressant. En effet, les images reconstruites sont pratiquement identiques et correspondent au maximum pixel-par-pixel de tous les exemples des données d'entraînements. Visuellement, chaque image décodée semble être les images d'apprentissage collées les unes sur les autres.

## 5. Convolution

Dans les années 1990, Yann Le Cun développe la technique des réseaux convolutifs. Nous avons souhaité étudier le fonctionnement de ce réseau en le confrontant à un problème de classification de 9 classes de chiffres manuscrits (MNIST) de taille **28x28x1**. Dans un premier temps, nous avons établi le réseau suivant :

**Conv2D** : Kernel : 7 x 7, Feature\_in : 1 (gray scale), Feature\_out : 16, Stride : 1

**Relu -> Maxpooling**: Stride : 2, Kernel : 2

**Conv2D** : Kernel : 3 x 3, Feature\_in : 16 (gray scale), Feature\_out : 28, Stride : 1

**Relu -> Maxpooling**: Stride : 2, Kernel : 2

**Linear** : feature\_in : 1008, feature\_out : 9

La fonction de perte utilisée est la **Softmax Cross Entropy** :

## SOFTMAX/CROSS-ENTROPY LOSS

- Consider a single training example  $x^{(0)}$  transformed as  
 $x^{(0)} \rightarrow z^{(1)} \rightarrow x^{(1)} \rightarrow \dots \rightarrow z^{(L)} \rightarrow x^{(L)}$
- The softmax function is  $x_i^{(L)} = p_i(z^{(L)}) = \frac{\exp(z_i^{(L)})}{\sum_{j=1}^{d_L} \exp(z_j^{(L)})}$
- The cross-entropy loss is  $J(z^{(L)}) = -\sum_{i=1}^{d_L} y_i \ln(p_i(z^{(L)}))$
- Gives us a notion of how good our classifier is

*Calcul de la softmax cross entropy*

### Hyperparamètres du réseau :

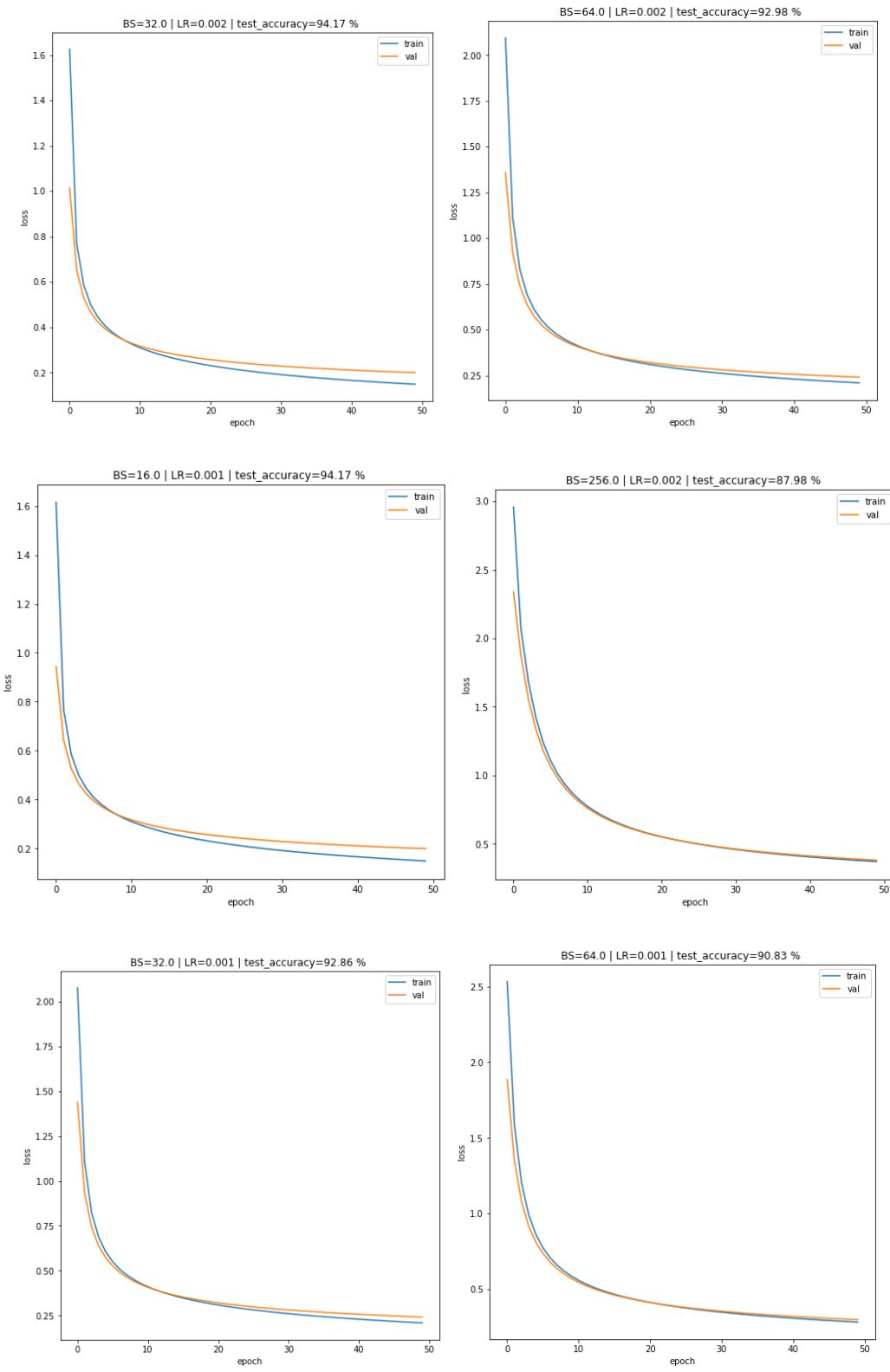
Pour le réseau précédent, nous avons souhaité déterminer la valeur en **accuracy de test** selon les hyperparamètres du modèle. Nous avons entraîné notre modèle sur le jeu MNIST, normalisé, avec un ensemble de train de 5040 images et 804 en test. Voici les résultats obtenus

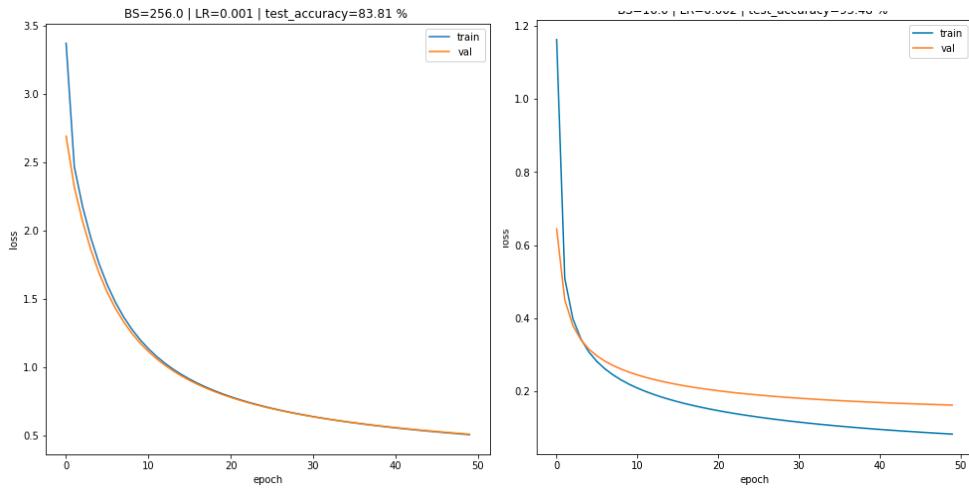
	Unnamed: 0	lr	batch_size	train_loss	val_loss	test_accuracy
0	0	0.002	16.0	[1.1622273370708234, 0.5089448233820123, 0.398...]	[0.6445830850062154, 0.4504215173585987, 0.379...]	0.954762
1	1	0.002	32.0	[1.6257777839453074, 0.7682442426539331, 0.585...]	[1.0142195640721403, 0.6511619851138601, 0.529...]	0.941667
4	4	0.001	16.0	[1.6148874072848252, 0.7668853260793717, 0.585...]	[0.9438731103398844, 0.6423137049278432, 0.527...]	0.941667
2	2	0.002	64.0	[2.0933869851238094, 1.1119322549294657, 0.827...]	[1.3577357810534505, 0.9196101393257359, 0.737...]	0.929762
5	5	0.001	32.0	[2.0759055598941916, 1.1067406049883168, 0.823...]	[1.4379209368668349, 0.9370646462955432, 0.743...]	0.928571
6	6	0.001	64.0	[2.5331205787890463, 1.5904673278906802, 1.205...]	[1.8850125873069792, 1.3551028940414682, 1.080...]	0.908333
3	3	0.002	256.0	[2.9566094749132428, 2.0707519964091605, 1.686...]	[2.338958285034268, 1.875279248723675, 1.55758...]	0.879762
7	7	0.001	256.0	[3.3716037867258413, 2.465599675215589, 2.1755...]	[2.690742801361969, 2.3165707974078185, 2.0668...]	0.838095

*Résultats selon hyperparamètres*

*Conv.ipynb cellule 7*

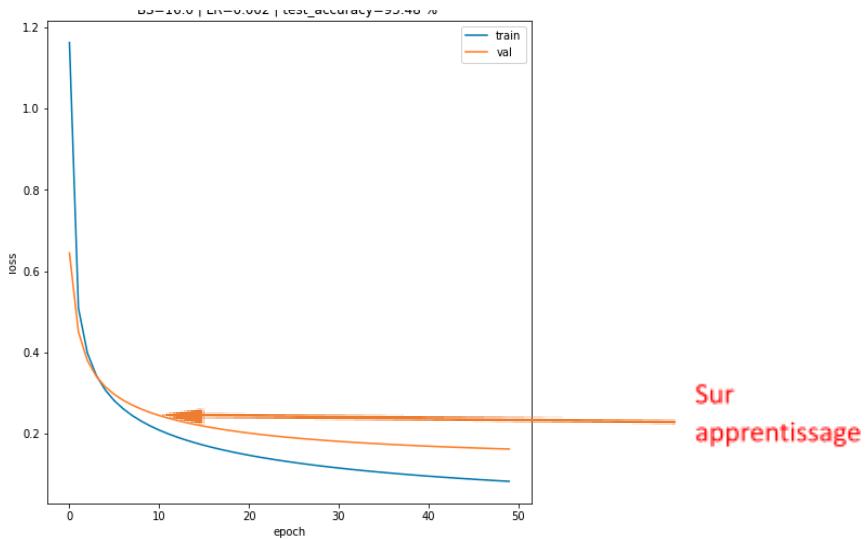
On visualise les courbes d'entraînements et de validations suivantes :





*Conv.ipynb cellule 9*

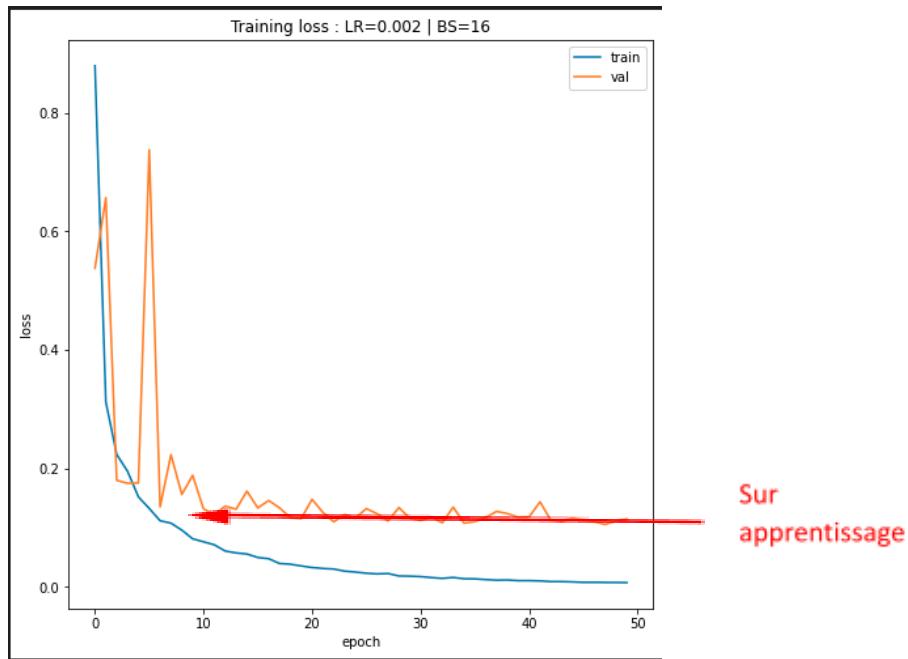
Pour une telle configuration en entraînement, nous observons de meilleures performances avec des petites tailles de lots. Il est important de préciser qu'à partir d'une certaine epoch, le modèle commence à sur apprendre :



*Exemple de sur apprentissage sur notre meilleure configuration*

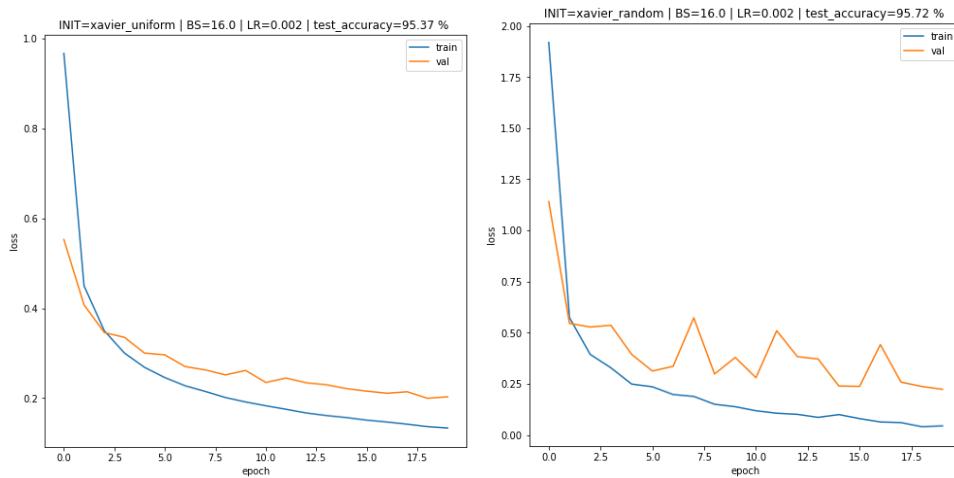
Nous verrons par la suite comment combattre ce sur apprentissage.

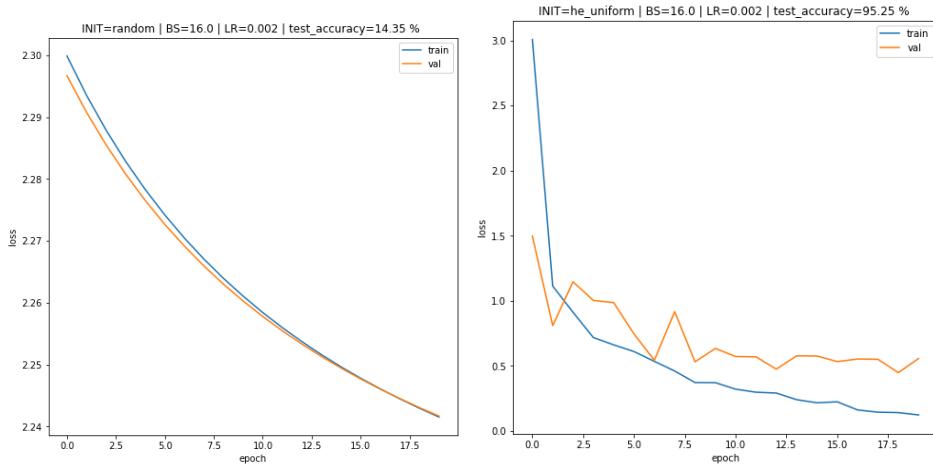
Que se passe-t-il quand nous augmentons notre nombre de données ? Nous sommes passés de 5040 images à 12600 pour entraîner notre modèle. Nous obtenons, sur un ensemble de test de 3240 images, **96.25 %**. Nous observons un sur apprentissage dès 5 epochs



## Initialisation et optimisation :

Nous avons étudié les performances de notre même réseau, sous différentes stratégies d'initialisation. Les résultats exposent la supériorité des méthodes dites de **xavier** (<https://paperswithcode.com/method/xavier-initialization>) et de **he** (<https://paperswithcode.com/method/he-initialization>). Voici les résultats :

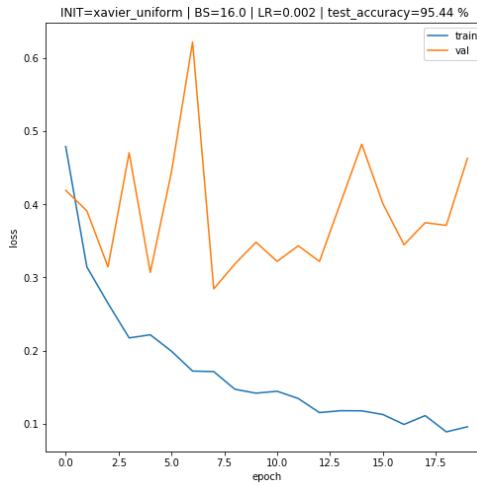




*He uniform*

L'initialisation par une constante (zéro par exemple), n'a pas fonctionné et n'a proposé que des pertes « nan ».

Concernant l'optimisation, nous avons opté pour une descente de gradient stochastique classique, et une optimisation **Adam** (<https://arxiv.org/abs/1412.6980>). Voici les résultats obtenus :

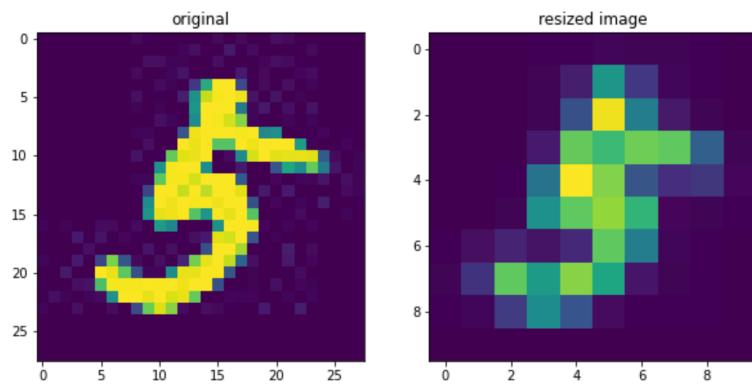


*Optimisation Adam*

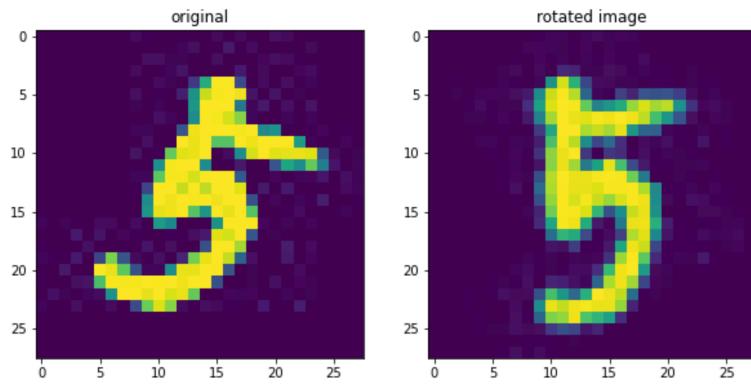
### Combattre le sur-apprentissage :

Dans cette partie, nous rendrons compte des expériences que nous avons mené concernant le sur-apprentissage.

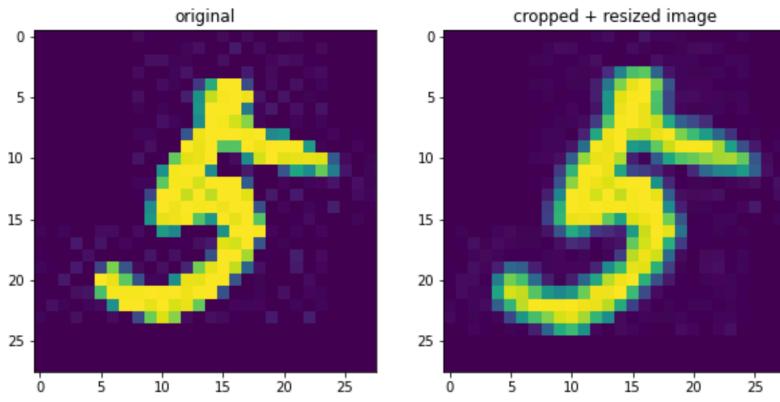
Dans un premier temps, nous avons utilisé la **data-augmentation**. Pour chaque point de données, nous donnons une probabilité de 0.3 d'être augmenté, cette probabilité est un hyperparamètre. Cette augmentation implique la rotation, la saturation et le recadrage.



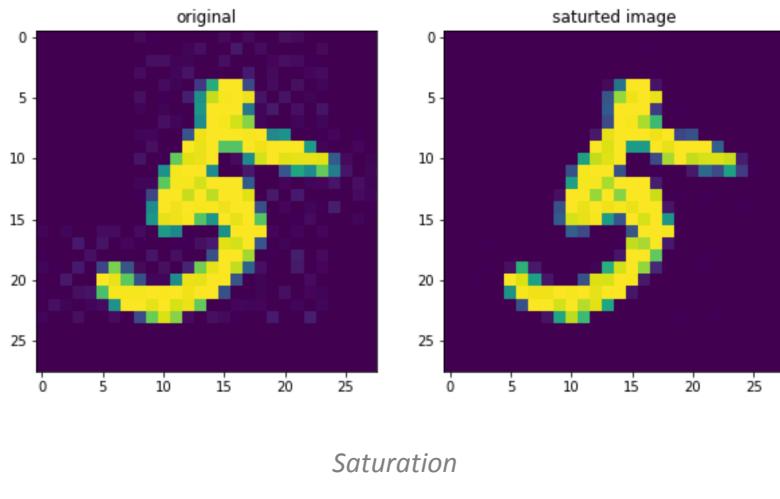
*Redimension*



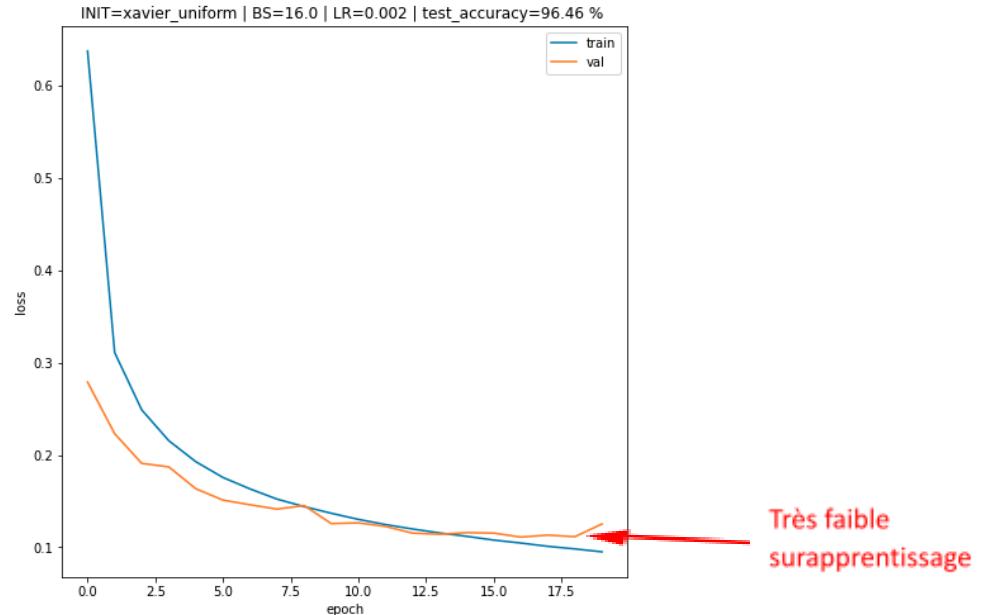
*Rotation*



*Recadrage*



Nous obtenons ainsi de bien meilleurs résultats, et une diminution du sur apprentissage. Les configurations sont les suivantes : learning rate : 0.002, batch size : 16, taille du jeu d'entraînement : 14400, taille du jeu de test : 1440. Nous observons une accuracy en validation s'élevant jusqu'à 97.6 %.



*Évolution du coût selon les epochs.*

Ces résultats sont très satisfaisants, d'autant plus que nous ne parvenions pas à obtenir de telles performances sur 50 epochs.

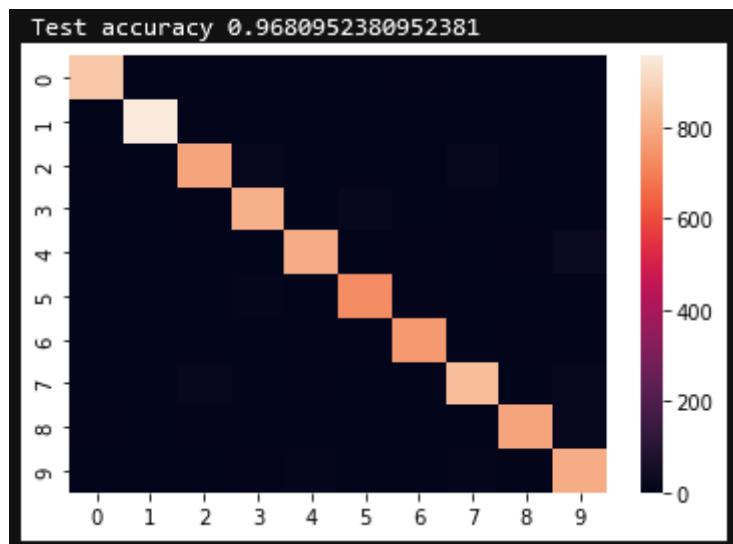
## 6. Interprétation d'un CNN

### 6.1. MNIST

Dans cette section, nous examinons les modèles de réseaux neuronaux convolutifs dans l'espoir de comprendre leur fonctionnement interne. Nous utilisons maintenant 33600 images MNIST comme ensemble d'entraînement, et 8400 images comme ensemble de test. L'architecture est la suivante.

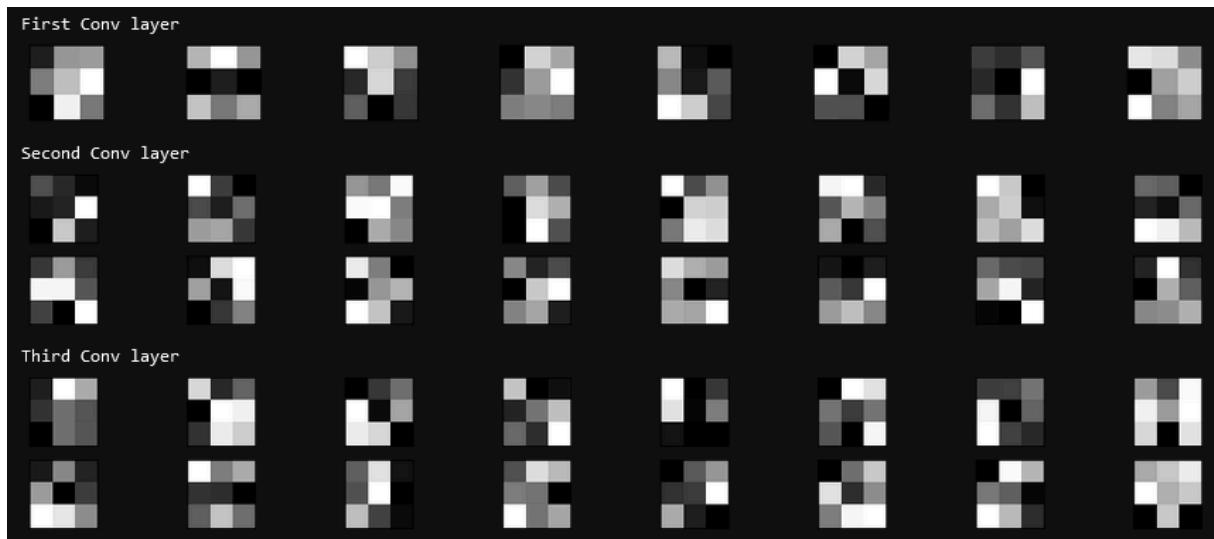
```
model = Sequential(  
    Conv2D(3, 1, 32, 1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Conv2D(3, 32, 64, 1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Conv2D(3, 64, 128, 1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Flatten(),  
    Linear(576*2, 64), ReLU(),  
    Linear(64, 10)  
)
```

#### 6.1.1. Performance

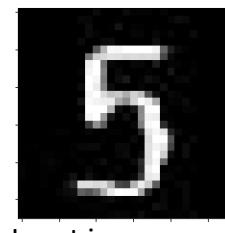


### 6.1.2. Visualisation des filtres

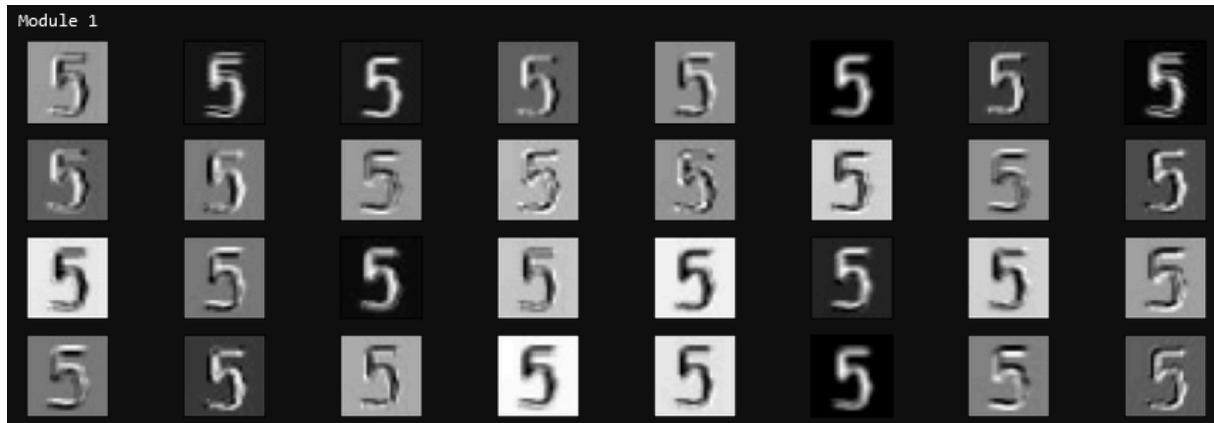
Nous voyons des motifs particuliers : point, ligne, gradient de luminosité, ... .



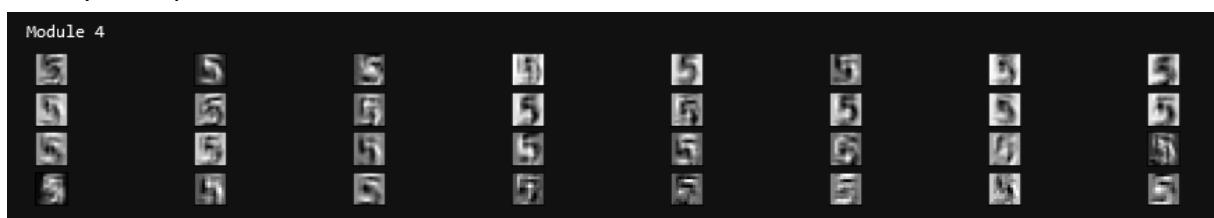
### 6.1.3. Feature maps



Input image



Nous voyons des filtres capturant de multiples informations: le trait de crayon, bord du trait de crayon, espace vide



Nous voyons des cartes de caractéristiques moins échantillonnées, avec moins de motifs à

haute fréquence.

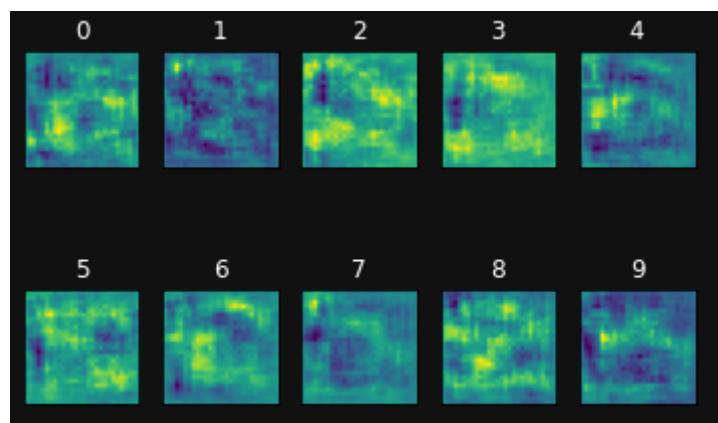


La carte des caractéristiques est encore sous-échantillonnée, afin de capturer les motifs de basse fréquence.

#### 6.1.4. Signal rétro propagé

Pour chaque classe, afin de visualiser ce que le modèle pense être un chiffre de la classe, nous traçons le signal rétro-propagation corigeant une image presque noire.

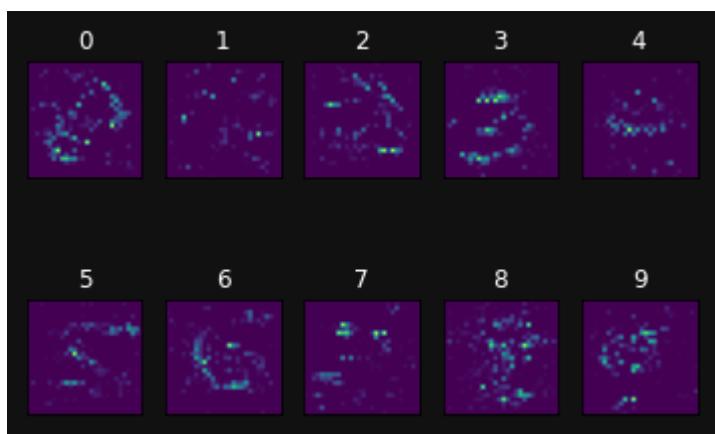
Nous avons constaté que le modèle n'apprend pas ce qu'est exactement un chiffre, à quoi il ressemble, comment il est écrit, mais plutôt les caractéristiques discriminantes d'un chiffre qui le rendent différent des autres, même si ce n'est qu'une partie du chiffre.



En raison de la saturation des fonctions d'activation, ce n'est pas une méthode de visualisation fiable.

#### 6.1.5. Visualisation des classes du modèle

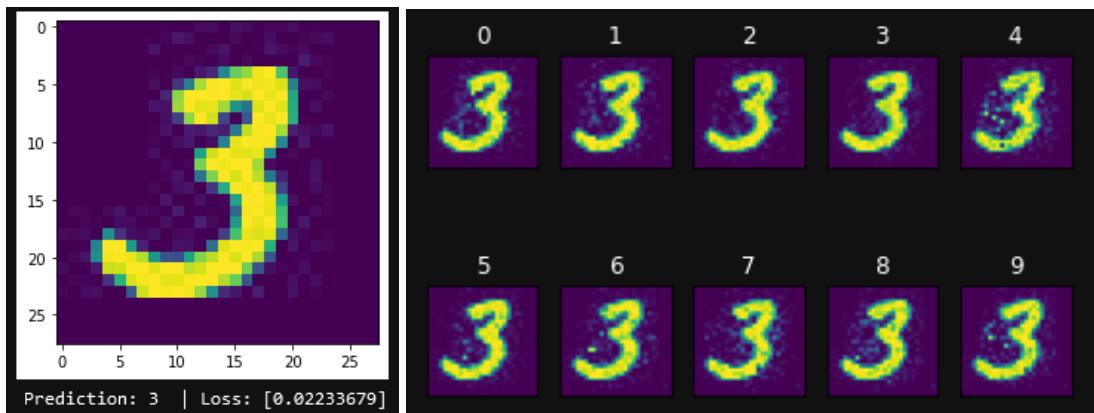
Nous essayons de visualiser ce que le modèle pense être une classe. En partant d'une image de bruit gaussien, nous définissons une fonction de perte comme la perte de classification de la classe correspondante, plus un terme de régularisation sur les pixels de l'image. Nous optimisons l'image en fonction de cette fonction de perte pour obtenir une image représentative de la classe.



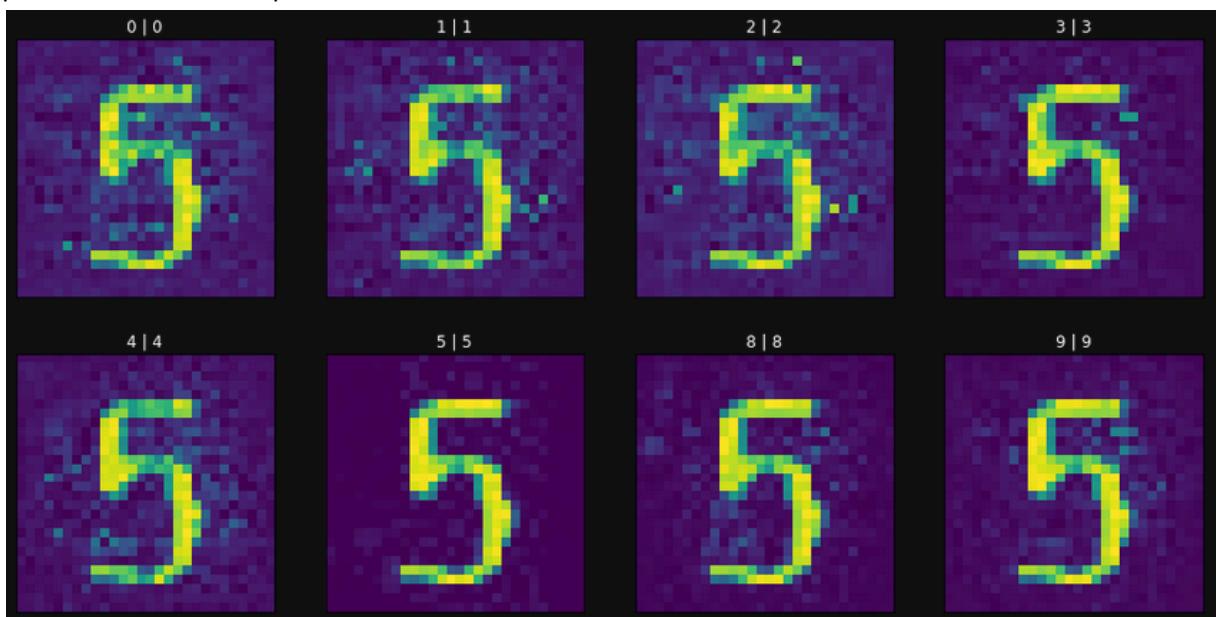
Comme les images numériques sont assez simples, nous obtenons une représentation raisonnable même avec une petite taille d'image de 32x32.

#### 6.1.6. Adversarial attack

Nous avons exploré l'attaque adversarial sur notre modèle. Étant donné un échantillon d'image de classe A que le modèle reconnaît correctement, nous générions une nouvelle image proche de l'image originale qui peut tromper le modèle en lui faisant croire qu'il s'agit de la classe B. En utilisant les paramètres fixes du modèle et de la classe cible comme entrée et vérité terrain, nous considérons l'image échantillonnée comme des paramètres à optimiser. Nous ajoutons un terme de régularisation à la perte de classification pour encourager l'image à ressembler à l'image originale.

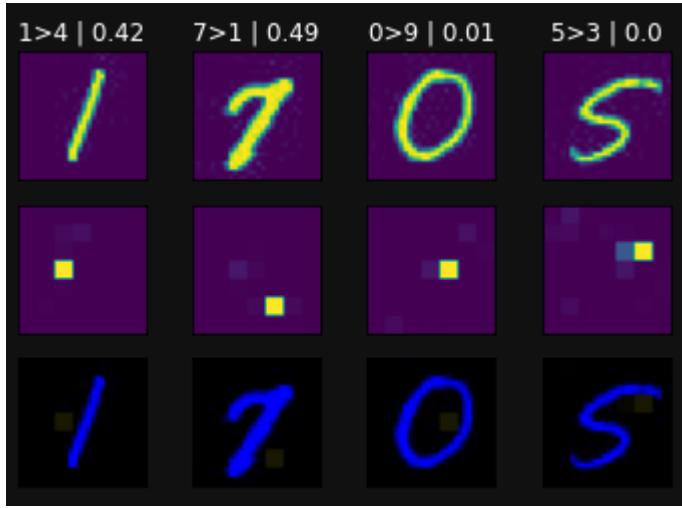


Nous tentons l'attaque sur un modèle plus fort (même architecture et entraînement, mais à une époque ultérieure). Nous voyons que le coût pour tromper le modèle est plus important, démontré par un niveau de bruit plus élevé.

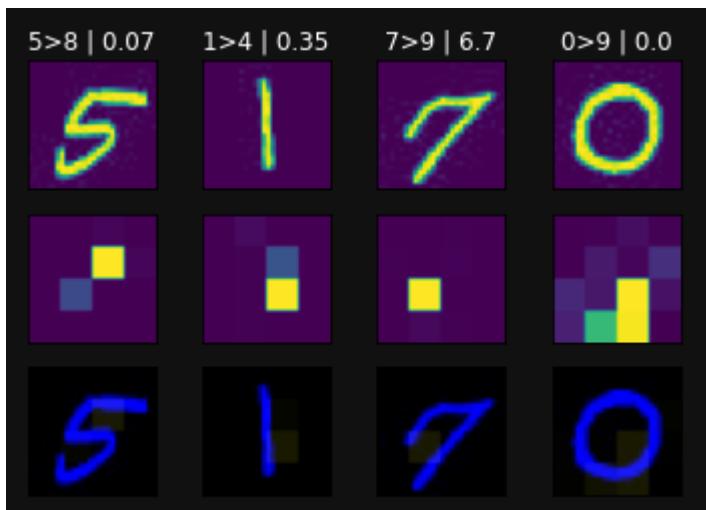


### 6.1.7. Test d'occlusion

Quelle parcelle de l'image est la plus importante pour que le modèle prenne sa décision ? Nous masquons des zones de l'image d'entrée avec du bruit et nous traçons l'augmentation de la perte de classification pour dessiner une carte thermique.



Nous voyons dans la première image qu'en regardant le point surligné, le modèle ne voit pas de triangle correspondant au chiffre 4, il écarte donc cette possibilité et décide qu'il s'agit du chiffre 1.



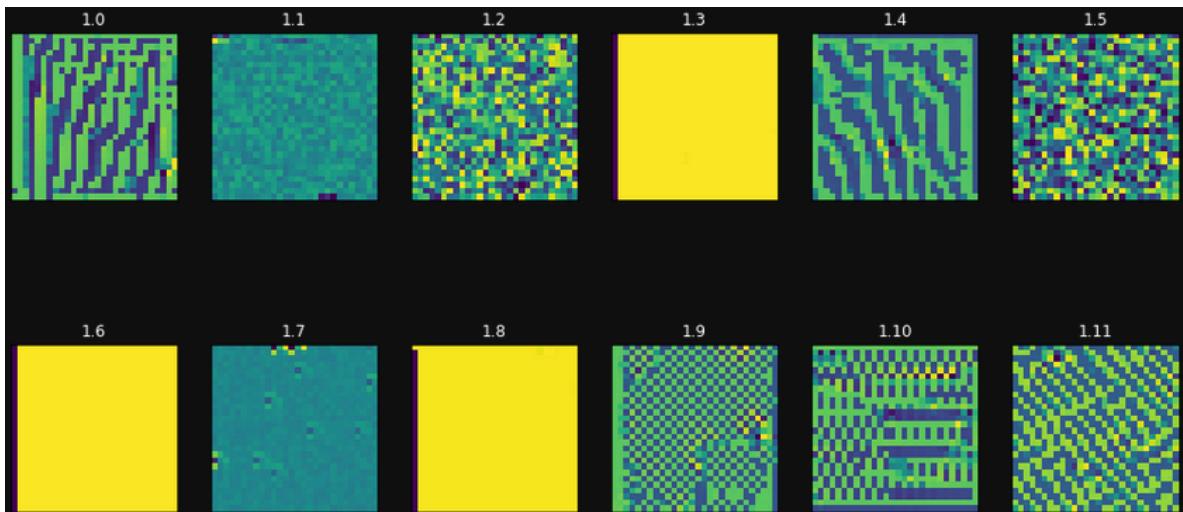
Dans la troisième image, l'absence de pixel lumineux dans la région en surbrillance empêche la formation d'un cercle complet, ce qui est nécessaire pour une prédiction de classe 9.

### 6.1.8. Visualisation de filtres par optimisation des features

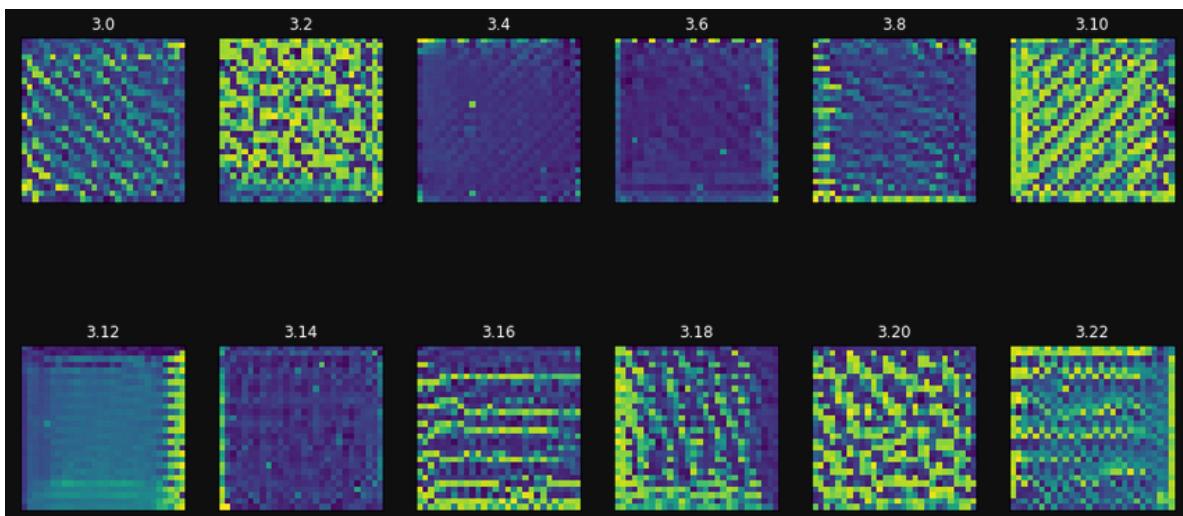
Nous optimisons la caractéristique d'entrée pour maximiser l'activation moyenne du filtre cible dans la couche convolutionnelle cible. Afin de préserver les caractéristiques de basse fréquence, nous optimisons de manière itérative une image à basse résolution et l'échantillonons à la taille de l'image cible. L'avantage de cette approche par rapport à l'optimisation d'une image à haute résolution à partir de zéro est que nous évitons d'être bloqués à un minimum local, et donc de ne pas représenter les caractéristiques à basse fréquence associées au filtre.

Nous affichons plusieurs filtres des trois couches convolutionnelles. Nous observons que les filtres des premières couches capturent des motifs simples, et que les couches ultérieures capturent des motifs plus complexes.

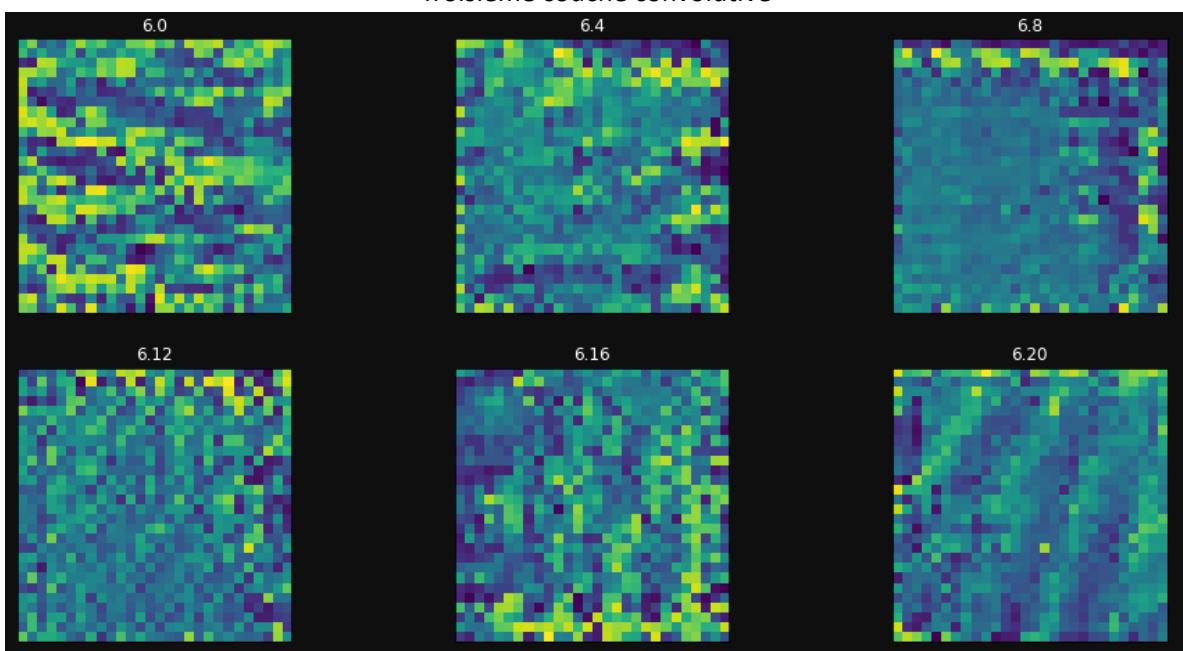
Première couche convulsive



Deuxième couche convulsive

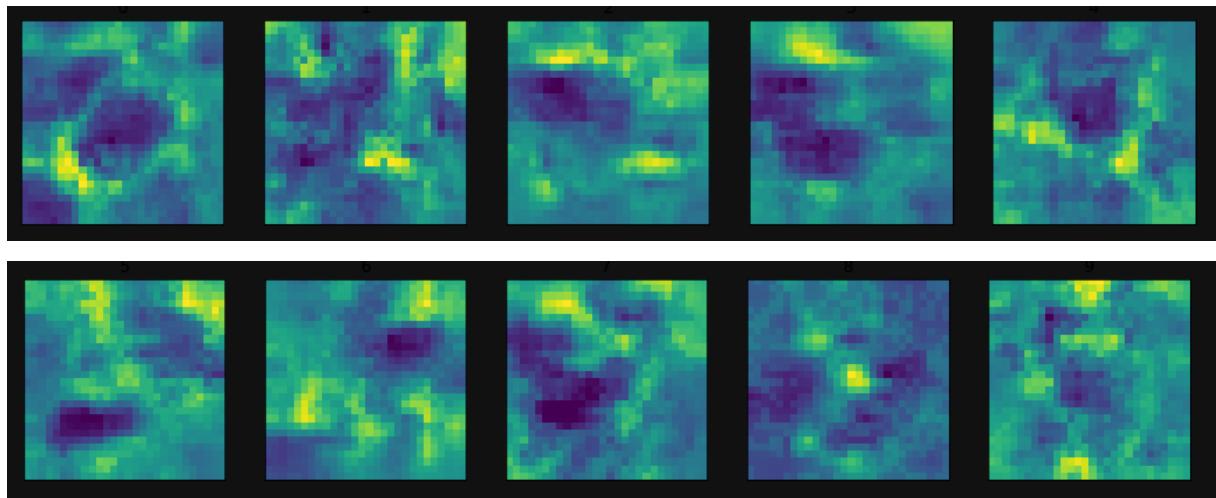


Troisième couche convulsive



### 6.1.9. Visualisation des classes avec un signal de basse fréquence

En intégrant la technique de 6.1.8. à 6.1.5., nous incluons les signaux de basse fréquence à la représentation de la classe, ce qui donne des visualisations encore meilleures.



## 6.2. CIFAR 10

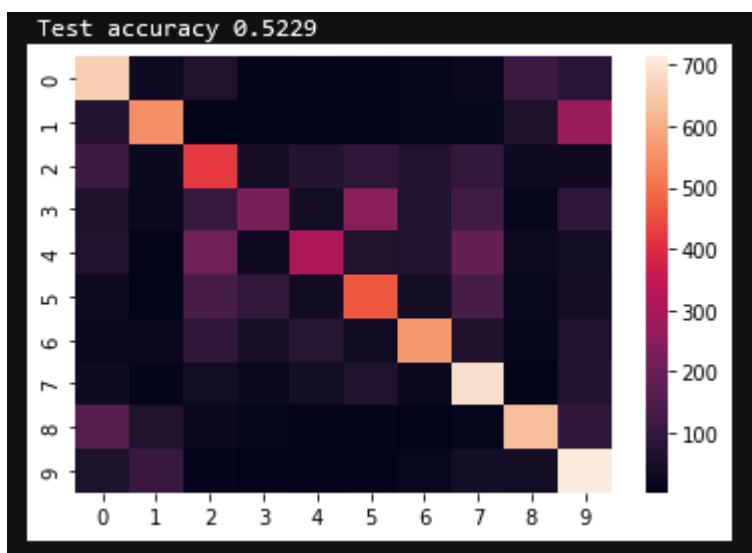
Nous étendons la tâche de classification d'images à un ensemble de données plus difficile : CIFAR10. Les images sont de taille 32x32 avec 3 canaux de couleur.

L'architecture du modèle est la suivante :

```
model = Sequential(  
    Conv2D(3, 3, 32, 1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Conv2D(3, 32, 64, 1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Conv2D(3, 64, 128, 1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Flatten(),  
    Linear(2048, 128), ReLU(),  
    Linear(128, len(label_names))  
)
```

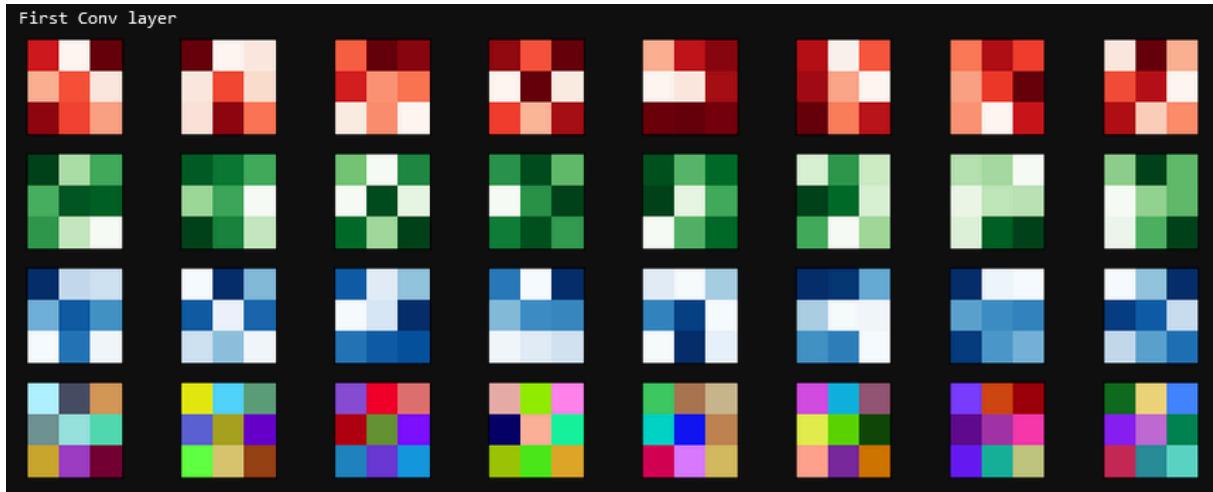
### 6.2.1. Performance

Le modèle peut distinguer les animaux des véhicules. Ce qui est difficile, c'est de faire la différence entre les animaux, et la différence entre les véhicules.

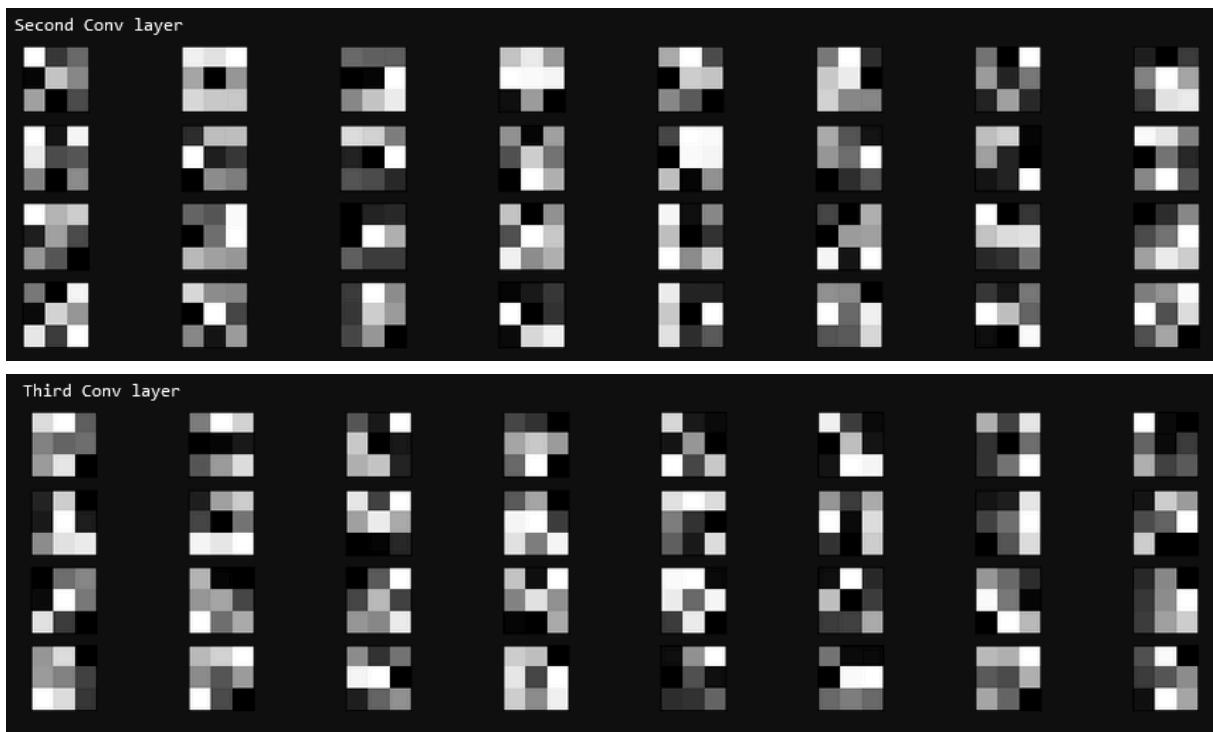


## 6.2.2. Visualize filters

Nous visualisons plusieurs noyaux de la première couche convulsive. Les lignes sont des noyaux s'appliquant respectivement sur le canal rouge, vert et bleu, suivis d'une combinaison des trois canaux. Ensuite, nous visualisons les noyaux de la deuxième couche convulsive. Chaque ligne correspond à un canal d'entrée, et chaque colonne correspond à un canal de sortie.



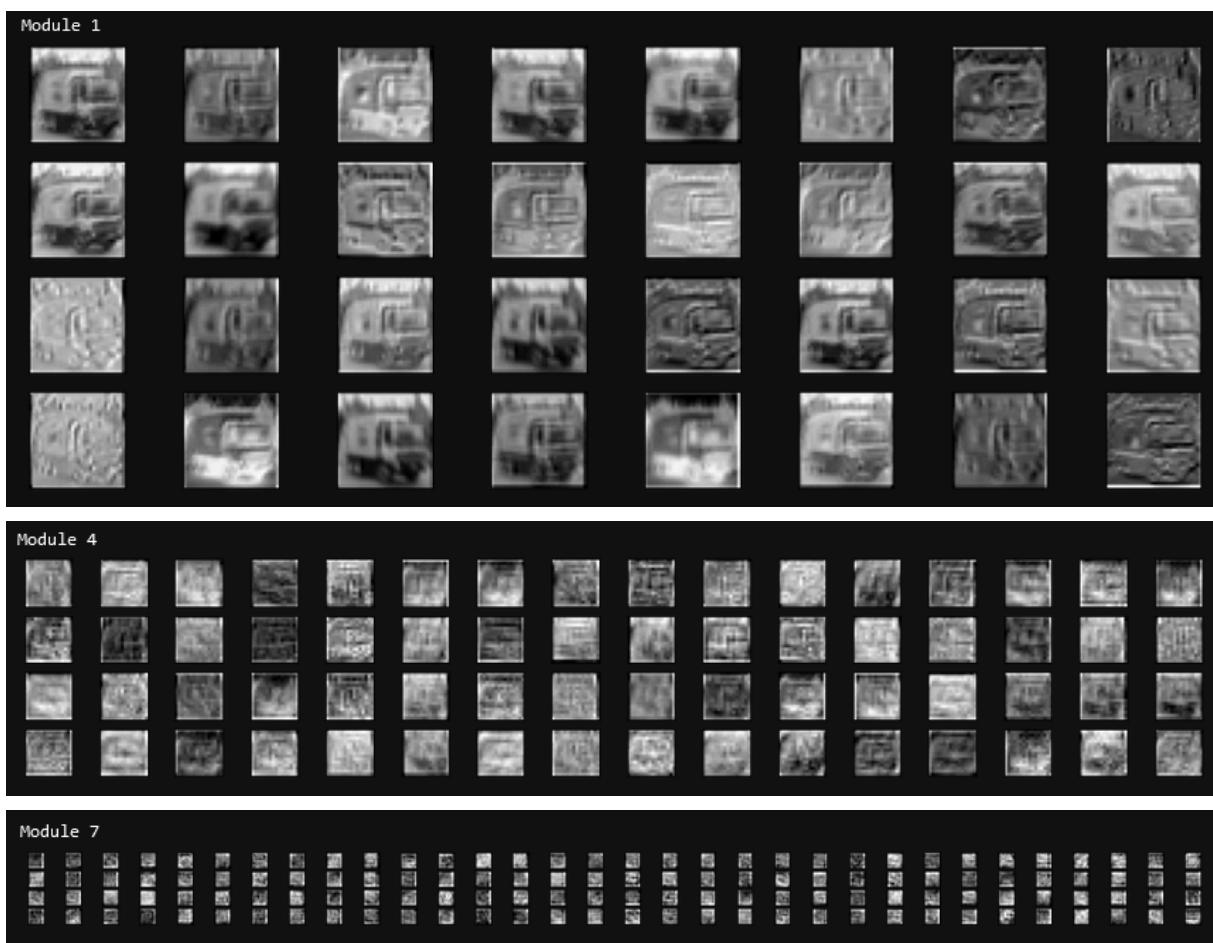
Certains filtres de la première couche semblent capturer des motifs spécifiques (ligne horizontale, coin) alors que d'autres semblent surchargés. Cela suggère qu'un modèle plus grand avec plus de canaux peut améliorer les performances.



Au niveau de la 2ème et 3ème couche convulsive, nous voyons encore des motifs intéressants : bord, point, gradient diagonal...

## 6.2.3. Feature maps

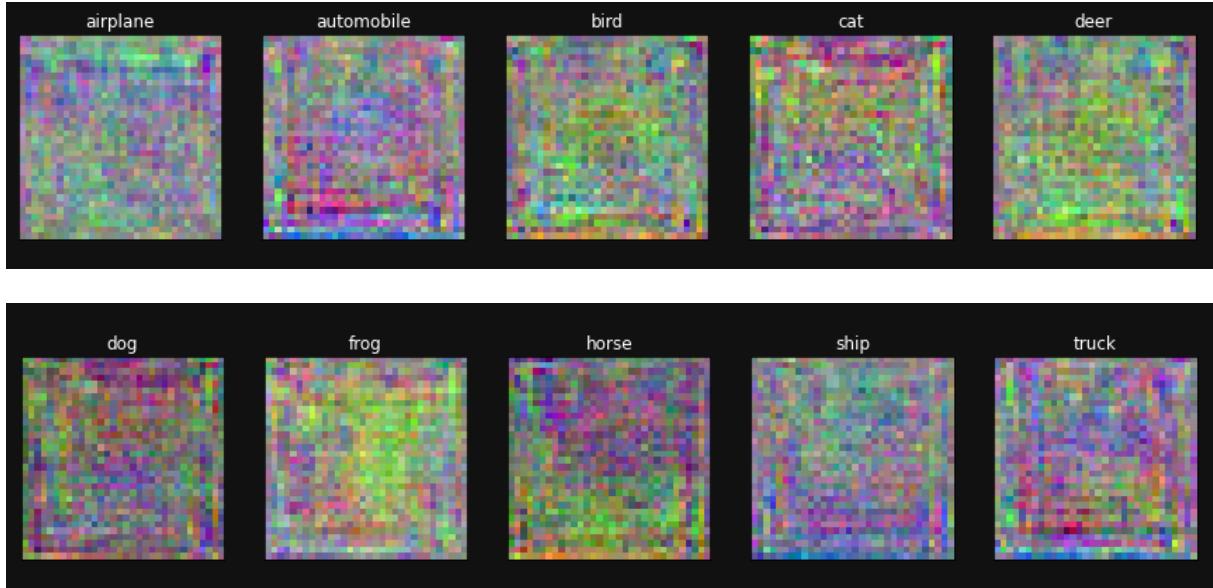
Au fur et à mesure que l'image est sous-échantillonnée, le modèle capture plus de signaux de basse fréquence et moins de signaux de haute fréquence.



#### 6.2.4. Backward signals

Nous visualisons également les signaux retro-propagation de plusieurs classes. Nous observons les détails intéressants suivants :

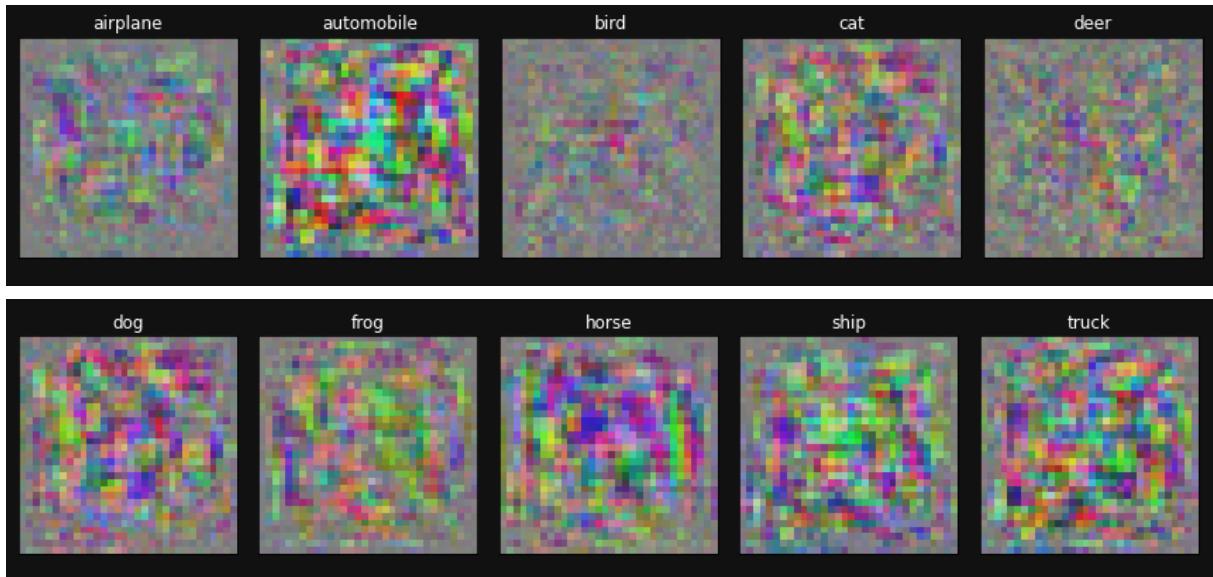
- Dans la classe "cheval", une tache rouge-brun au-dessus d'une tache verte, ressemblant à un corps de cheval sur un champ d'herbe
- Dans "oiseau", "cerf" et "grenouille", de nombreux pixels verts, probablement en raison de l'environnement dans lequel vivent les cerfs et les grenouilles.



#### 6.2.5. Visualize class model

Contrairement aux images MNIST, les images CIFAR-10 sont plus complexes, et la taille de l'image 32x32 n'est pas suffisante pour produire des bonnes représentatives des classes.

On constate que les véhicules présentent de nombreux motifs horizontaux et verticaux, et que les voitures sont colorées. Les oiseaux et les cerfs sont caractérisés par des motifs à haute fréquence (plumes et bois), contrairement aux chiens, aux chats et aux chevaux.



## 6.2.6. Adversarial attack

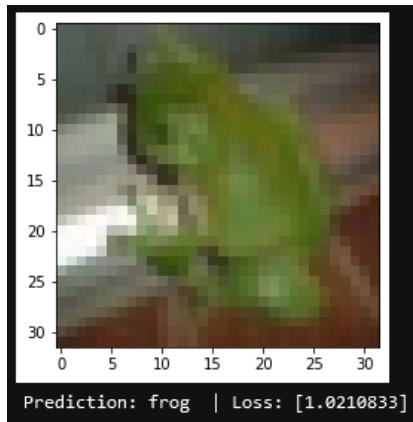


Image originale. Cette image est bien classée par le modèle comme une grenouille, avec une valeur de perte de 1,02.

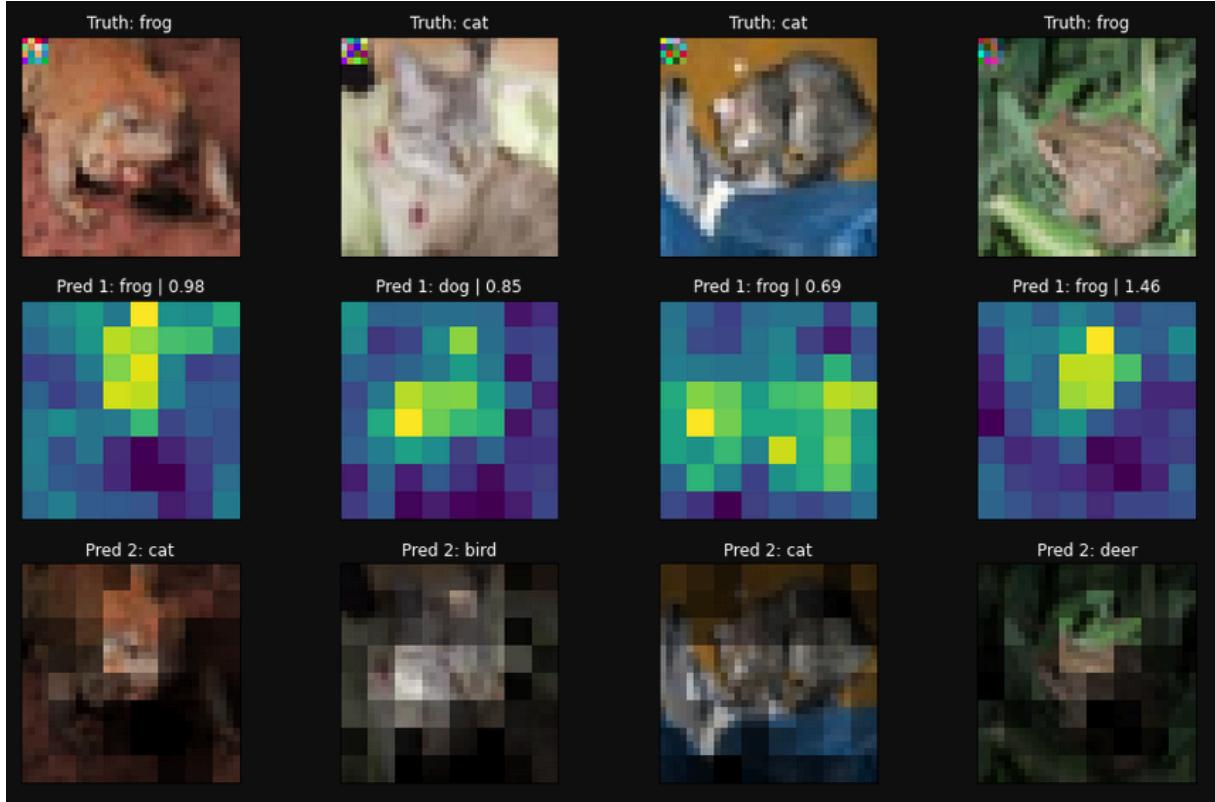


Image générée que le modèle reconnaît comme un camion. La valeur de perte pour cette classification est de 2,06, une bonne performance avec seulement 6 pixels visuellement modifiés.

Pour chaque classe, nous pouvons générer un exemple contradictoire capable de tromper le modèle.



### 6.2.7. Occlusion test



```

airplane>ship (0.19)
airplane>ship (0.09)
ship>automobile (0.37)
ship>airplane (0.09)

```



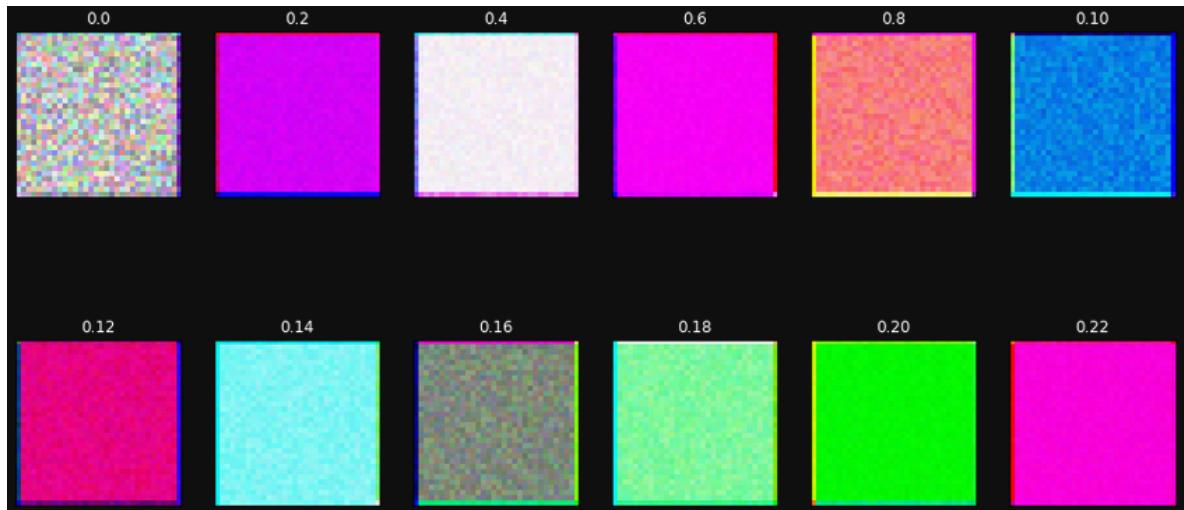
On voit sur la deuxième image que le modèle a regardé l'avion. La partie surlignée en dessous était homogène avec l'arrière-plan, ce qui suggère qu'il n'y a pas d'eau, et qu'il ne peut donc pas s'agir d'un navire, ce qui est la deuxième hypothèse. L'objet doit planer dans l'air, c'est donc un avion.

Dans la troisième image, la partie surlignée se trouve juste en dessous du bateau, où l'on peut voir son reflet. Il doit donc être sur l'eau. L'automobile n'est jamais sur l'eau, donc la classe bateau l'emporte sur ce candidat.

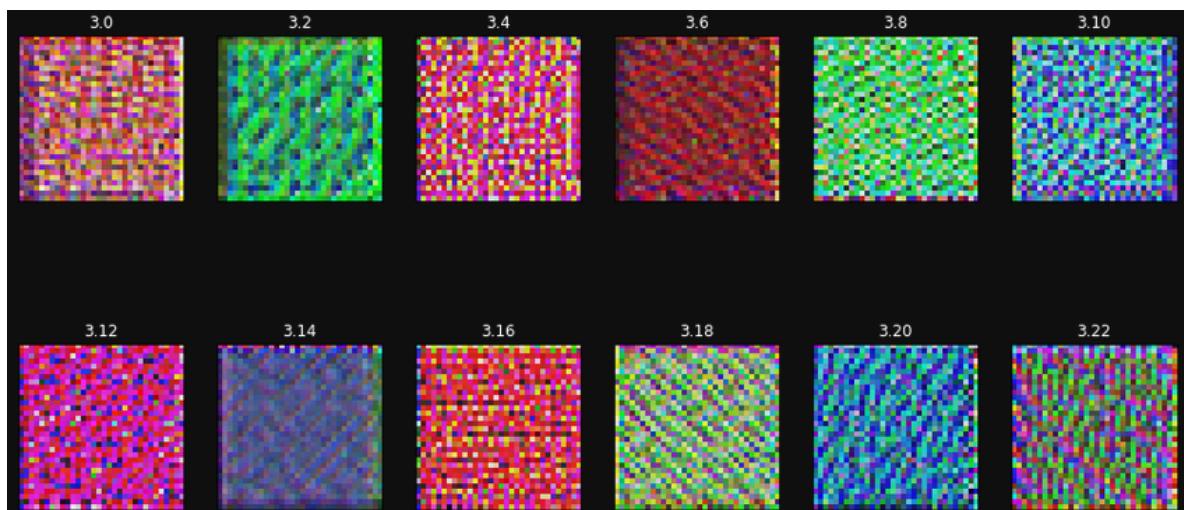
### 6.2.8. Filter visualization by Optimizing features

La première couche capte les signaux à haute fréquence comme la texture ou les couleurs. La deuxième capture davantage de motifs, et cela devient plus complexe dans la troisième et la quatrième.

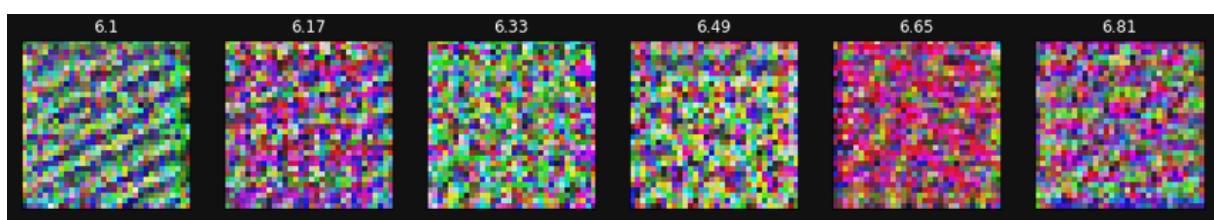
Première couche convulsive



Deuxième couche convulsive



Troisième couche convulsive



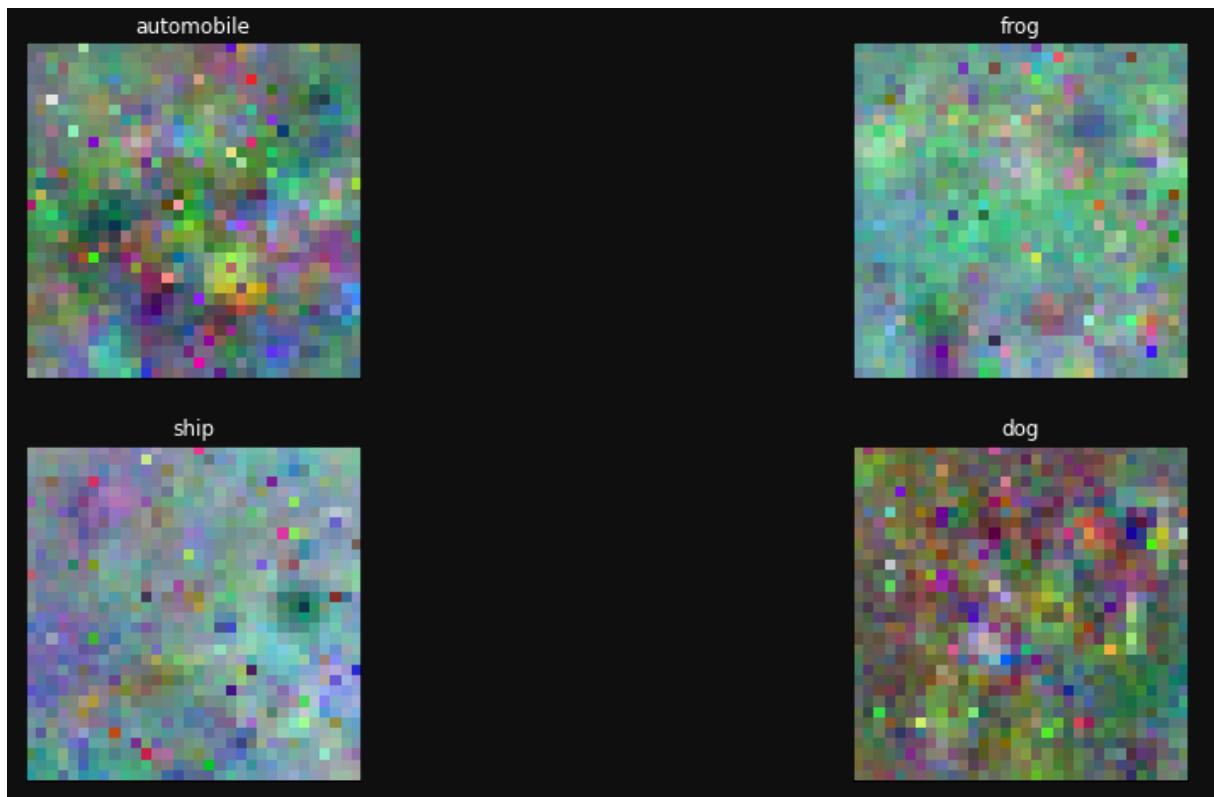
### 6.2.9. Visualisation des features par classe

En raison de la petite taille de l'image, la visualisation n'est pas tout à fait satisfaisante. Nous pouvons observer des détails qui sont cohérents à travers les différentes ré-exécutions.



#### 6.2.9.2. Optimisation : classification loss

Au lieu de se concentrer sur le logit de la classe, cette optimisation essaie de réduire les logits des autres classes. Ce n'est pas très utile pour comprendre la classe ciblée.



### 6.3. FRUITS-360

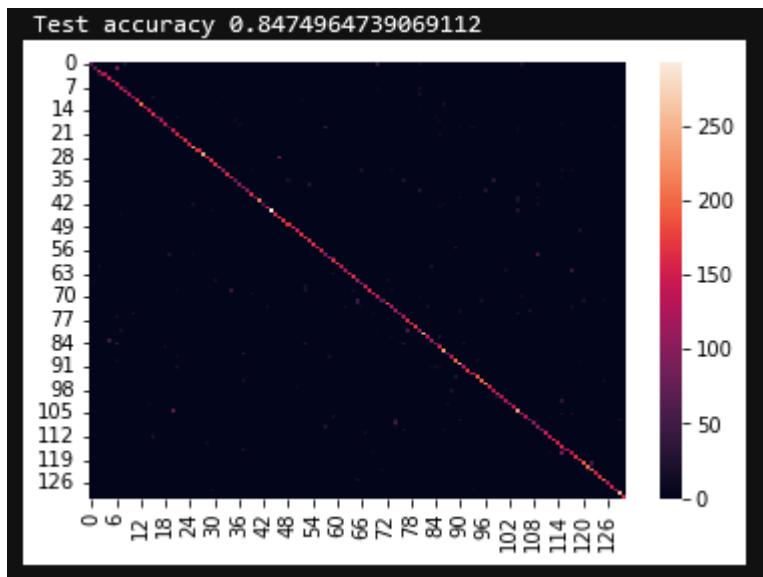
Fruits-360 dataset: images de fruits de taille (100x100x3). Il y a 131 classes de fruits (67k exemples de formation, 23k exemples de test).



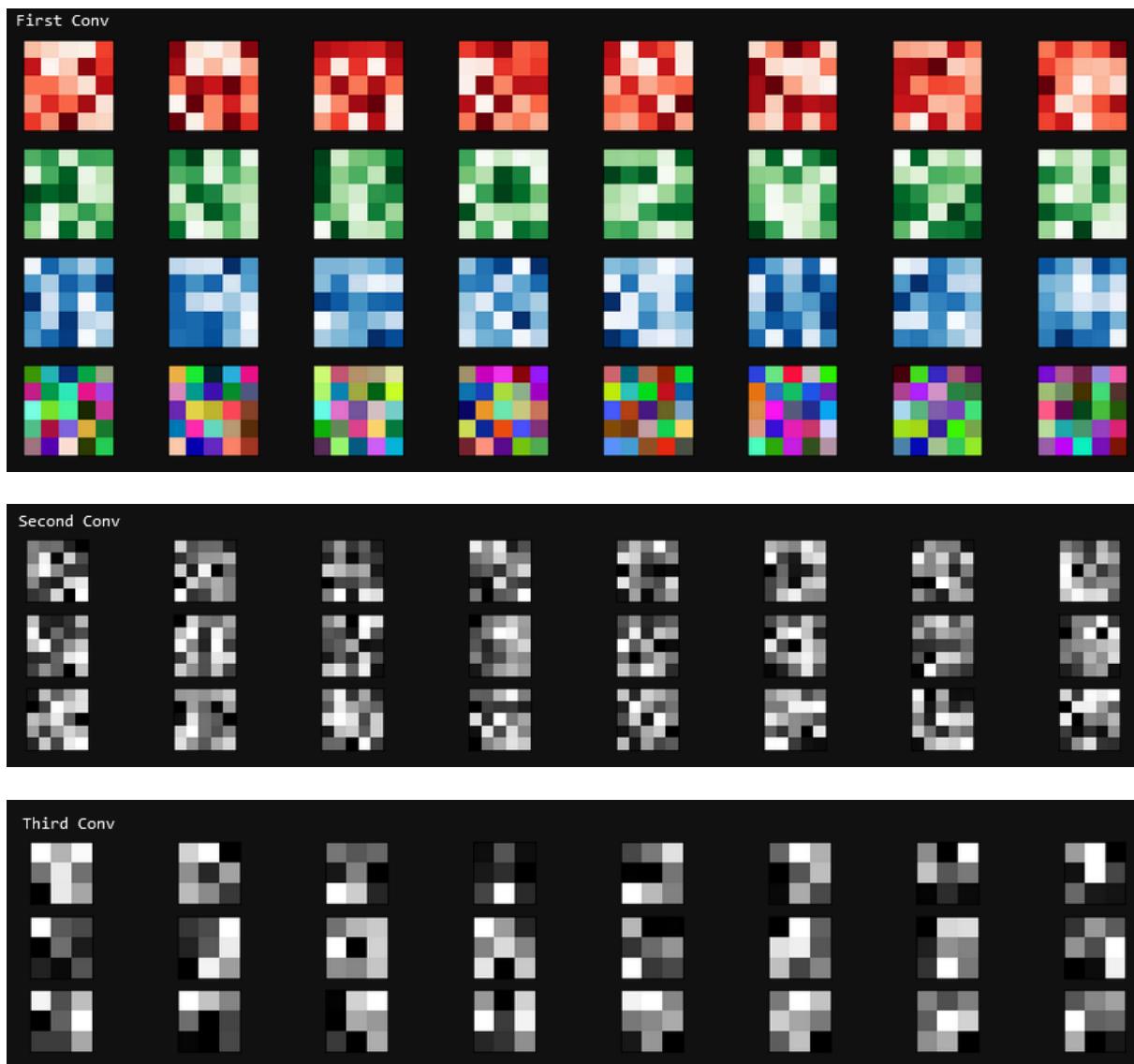
Model architecture

```
model = Sequential(  
    Conv2D(5, 3, 32, stride=2, padding=2, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 1),  
    Conv2D(5, 32, 64, stride=2, padding=3, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 1),  
    Conv2D(3, 64, 128, stride=1, padding=1, init="xavier_uniform", bias='zero'), ReLU(),  
    MaxPool2D(2, 2, 0),  
    Flatten(),  
    Linear(2048, 128), ReLU(),  
    Linear(128, len(labels))  
)
```

6.3.1. Performance

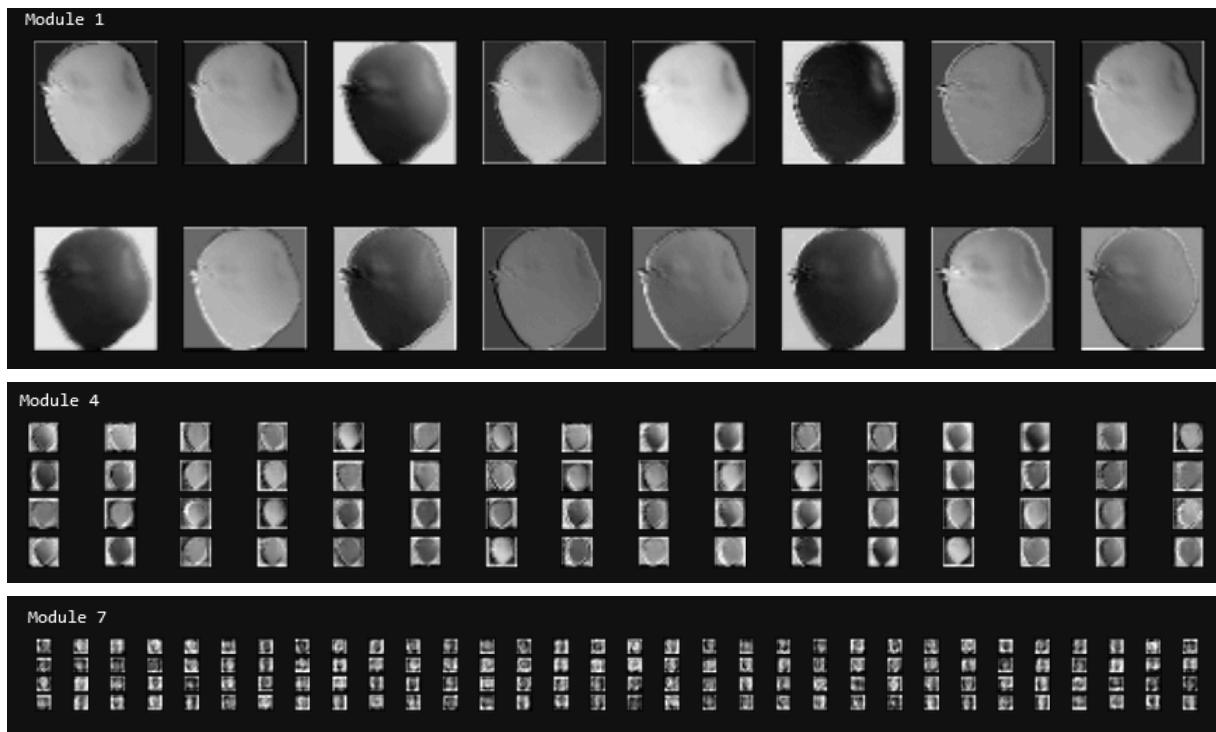


### 6.3.2 Visualize filters

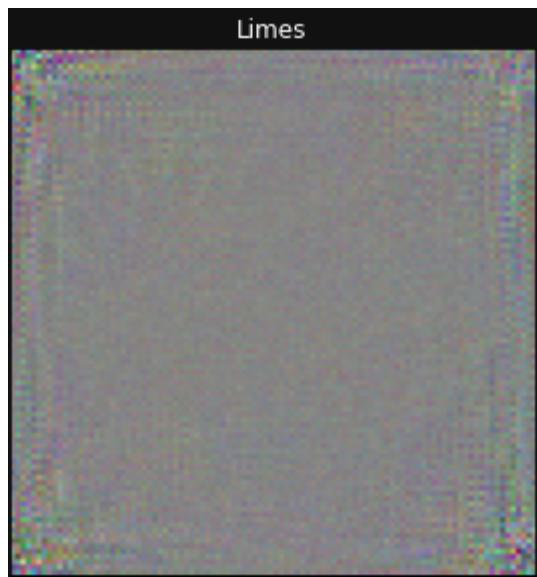
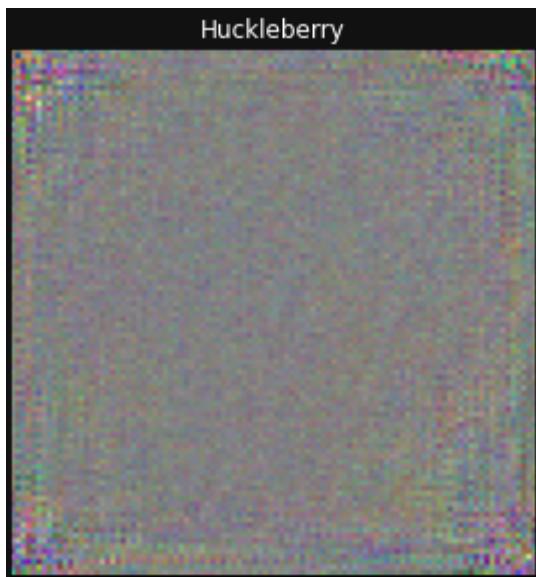


### 6.3.3. Visualize feature maps

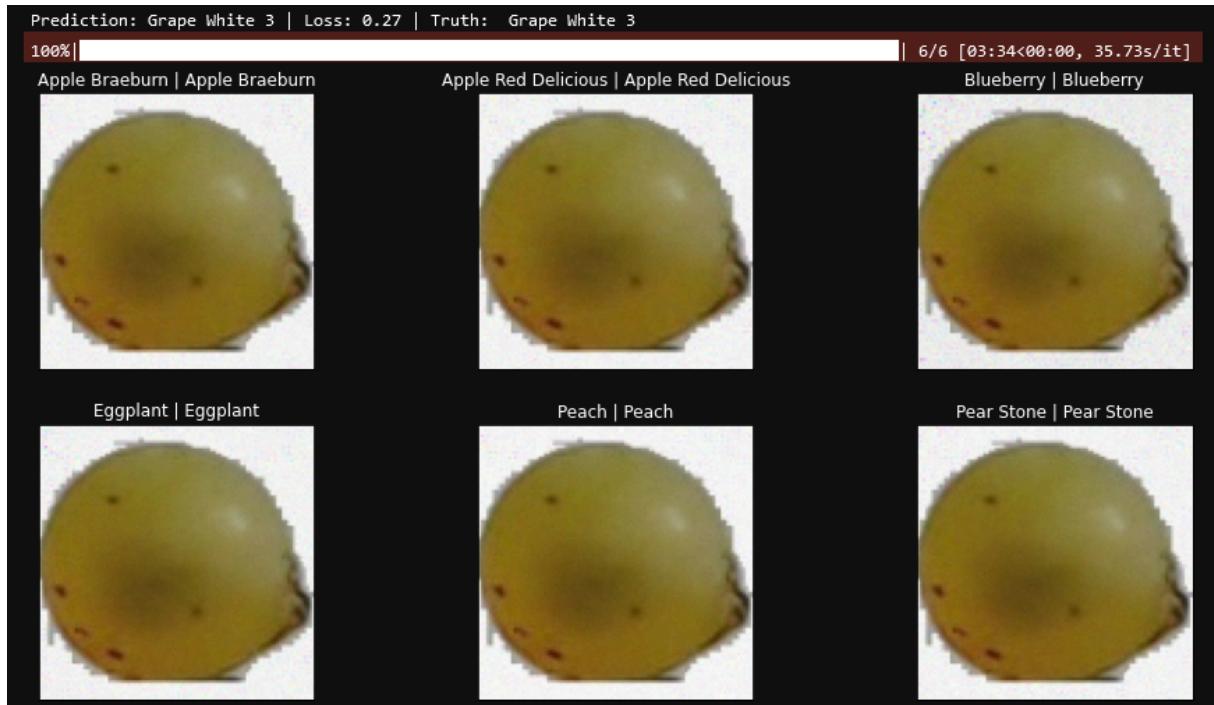




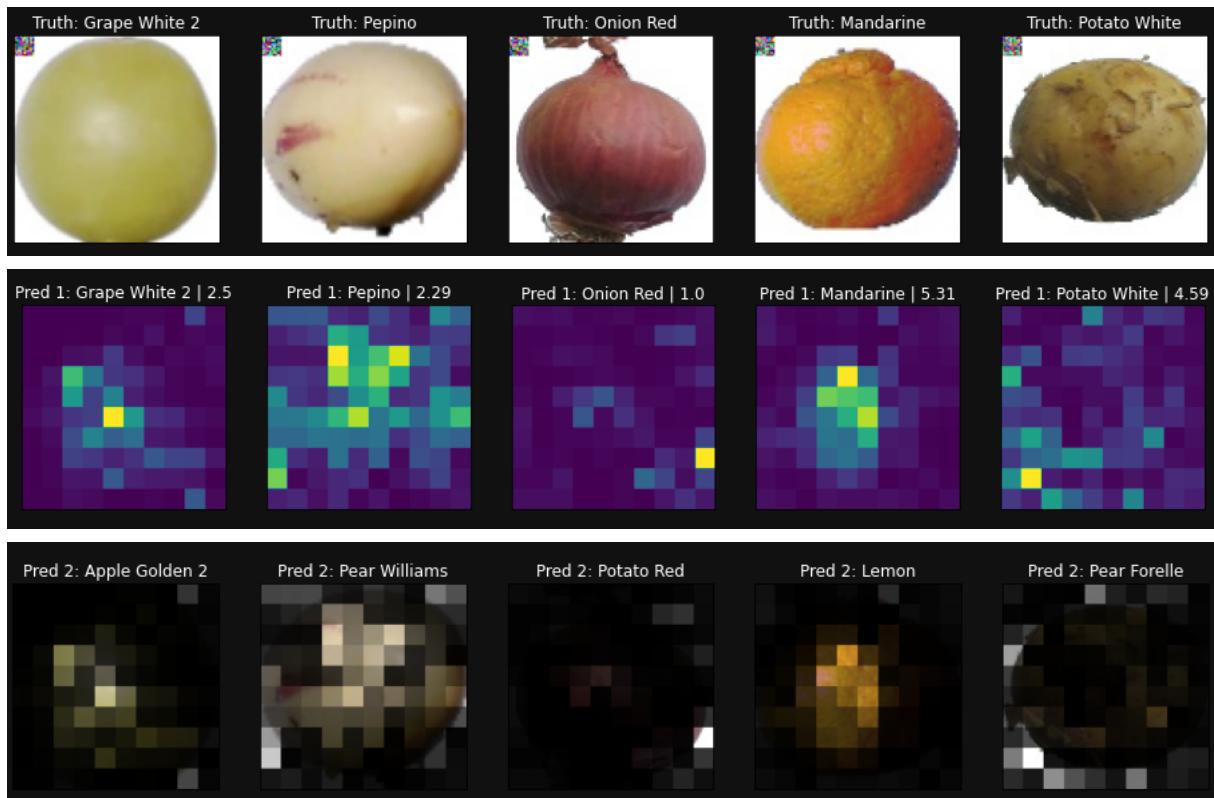
#### 6.3.4. Visualize backward signals



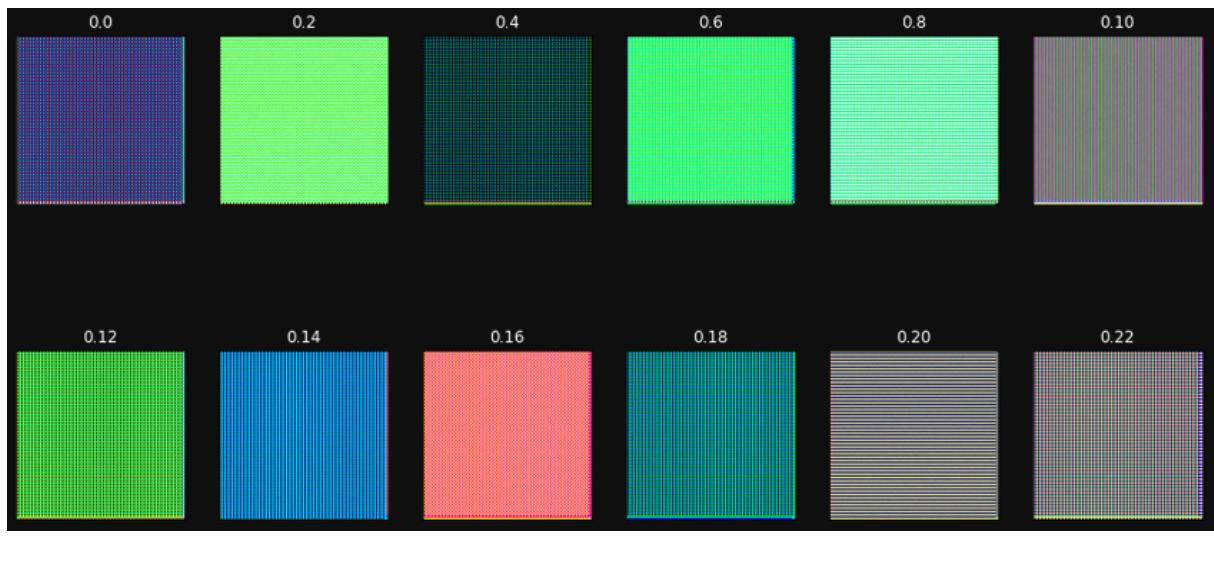
### 6.3.5. Adversarial attack



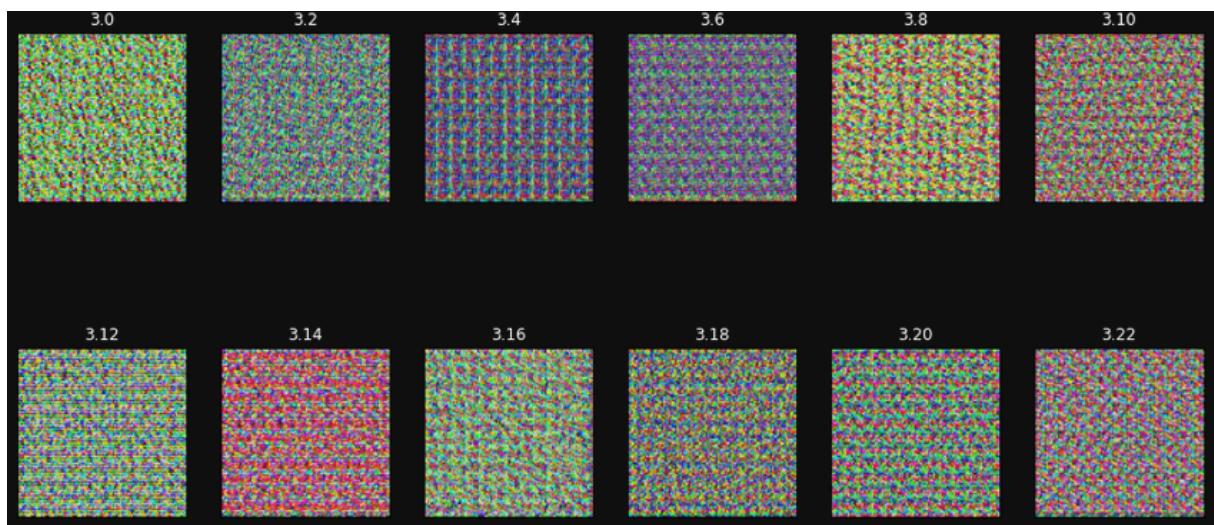
### 6.3.6. Occlusion test



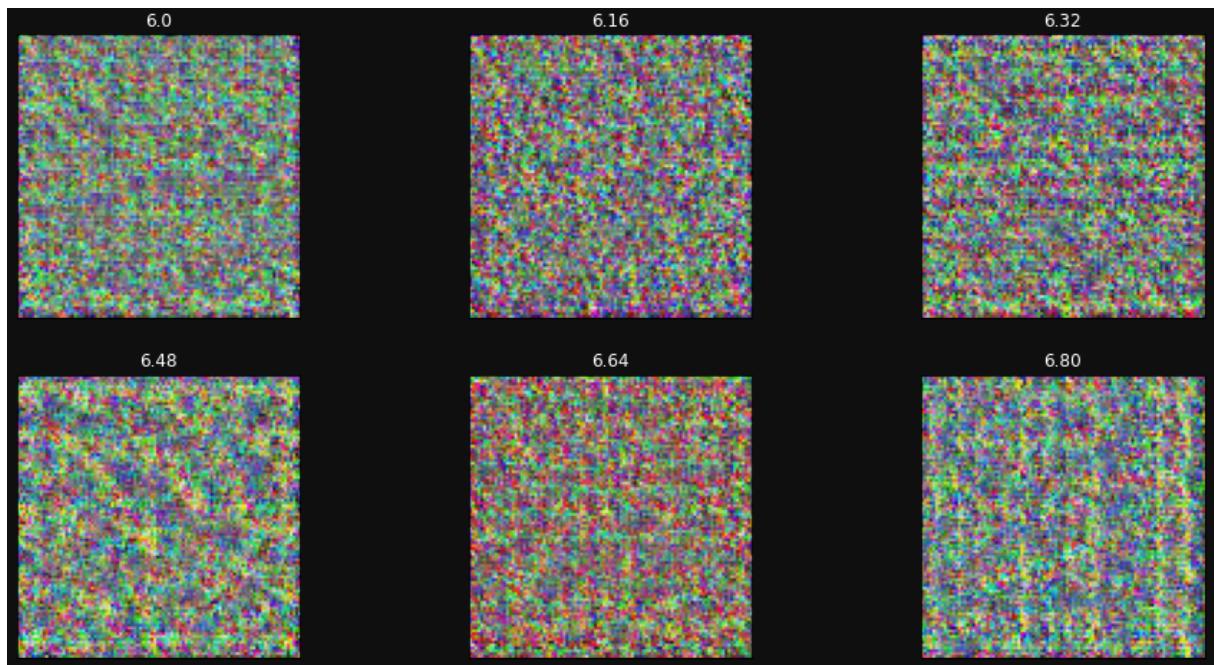
### 6.3.7. Feature visualization - Filters



Conv layer 1



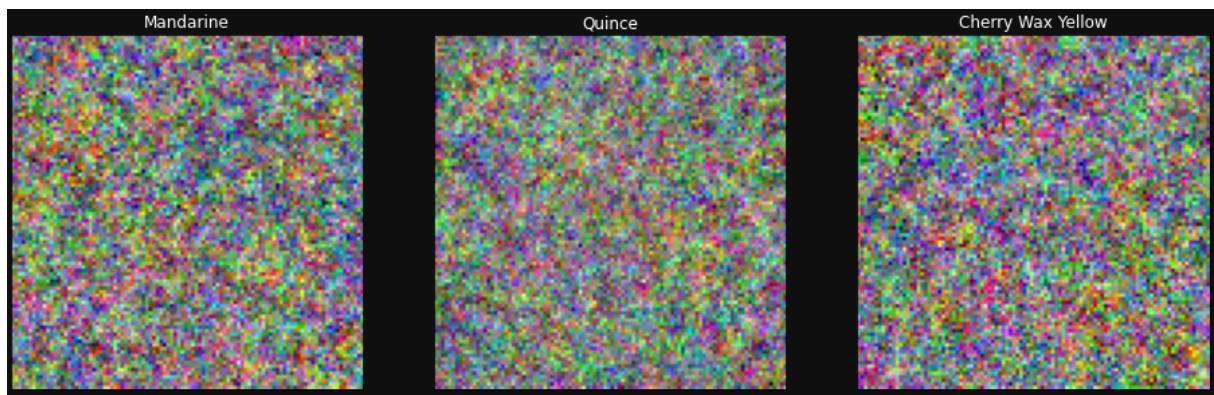
Conv layer 2



Conv layer 3

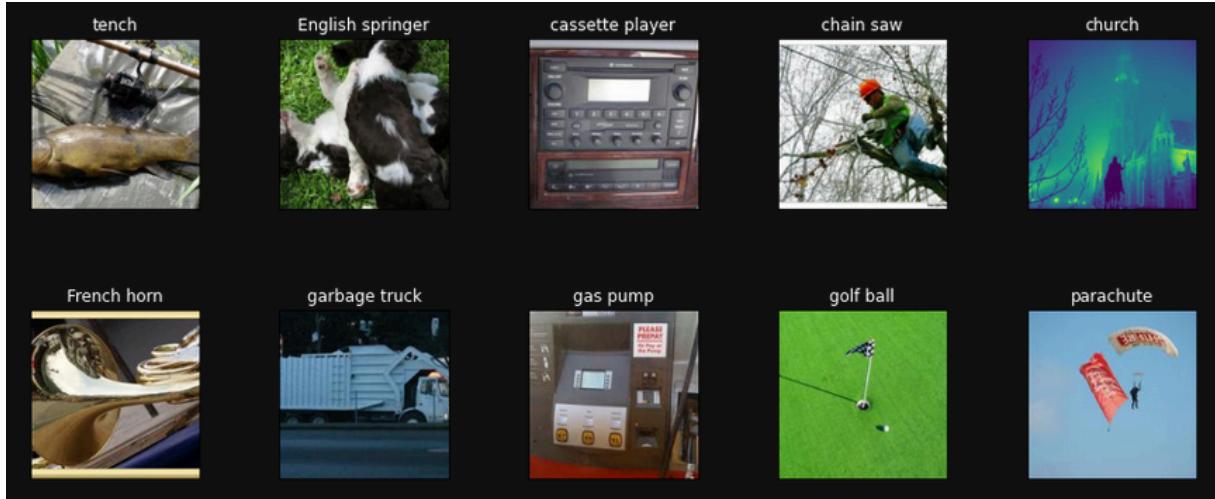
### 6.3.8. Feature visualization - Class

#### 6.3.8.1. Logit optimization



## 6.4. Imagenette

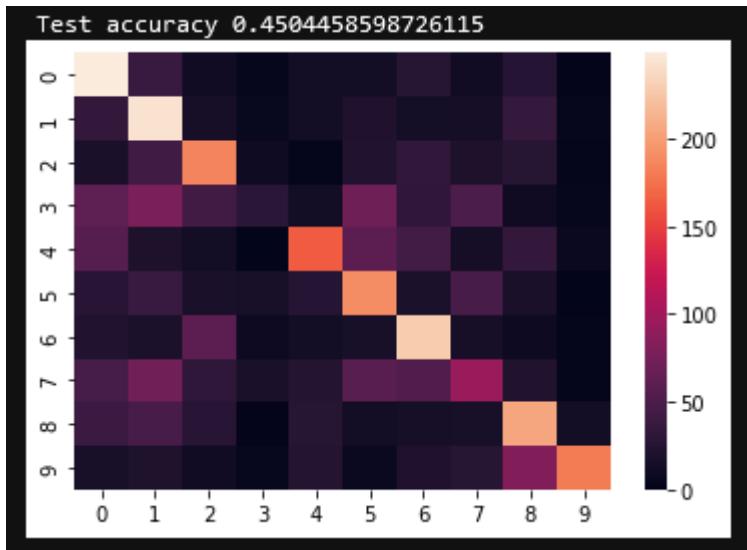
Imagenette dataset: Taille de l'image (160,160,3). 9469 exemples de formation, 3925 exemples de test.



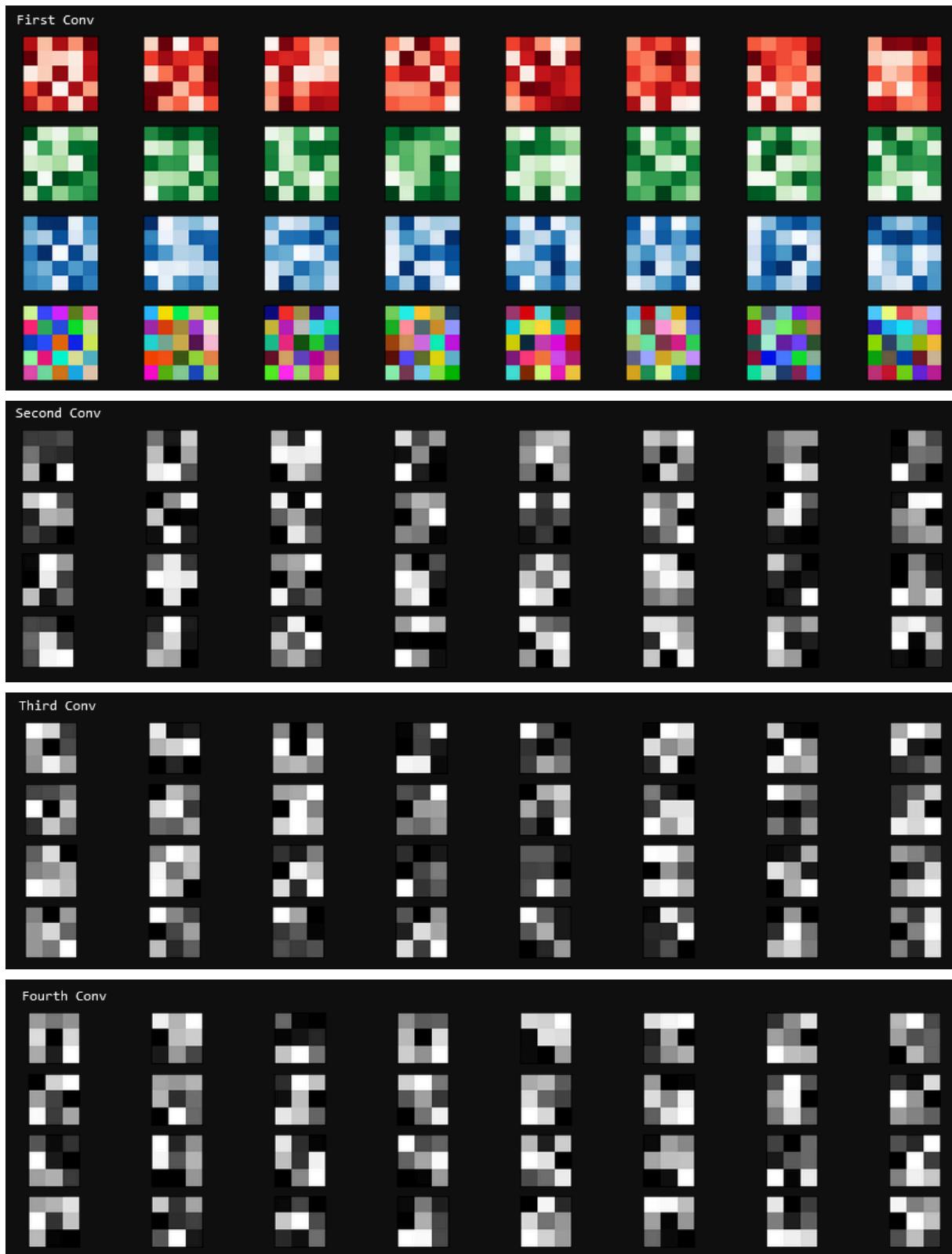
Model architecture

```
model = Sequential(  
    Conv2D(5, 3, 32, stride=2, padding=2, init="xavier_uniform", bias='zero'), ReLU(), MaxPool2D(2, 2, 0),  
    Conv2D(3, 32, 32, stride=1, padding=1, init="xavier_uniform", bias='zero'), ReLU(), MaxPool2D(2, 2, 0),  
    Conv2D(3, 32, 64, stride=1, padding=1, init="xavier_uniform", bias='zero'), ReLU(), MaxPool2D(2, 2, 0),  
    Conv2D(3, 64, 128, stride=1, padding=1, init="xavier_uniform", bias='zero'), ReLU(), MaxPool2D(2, 2, 0),  
    Flatten(),  
    Linear(3200, 128), ReLU(),  
    Linear(128, len(labels))  
)
```

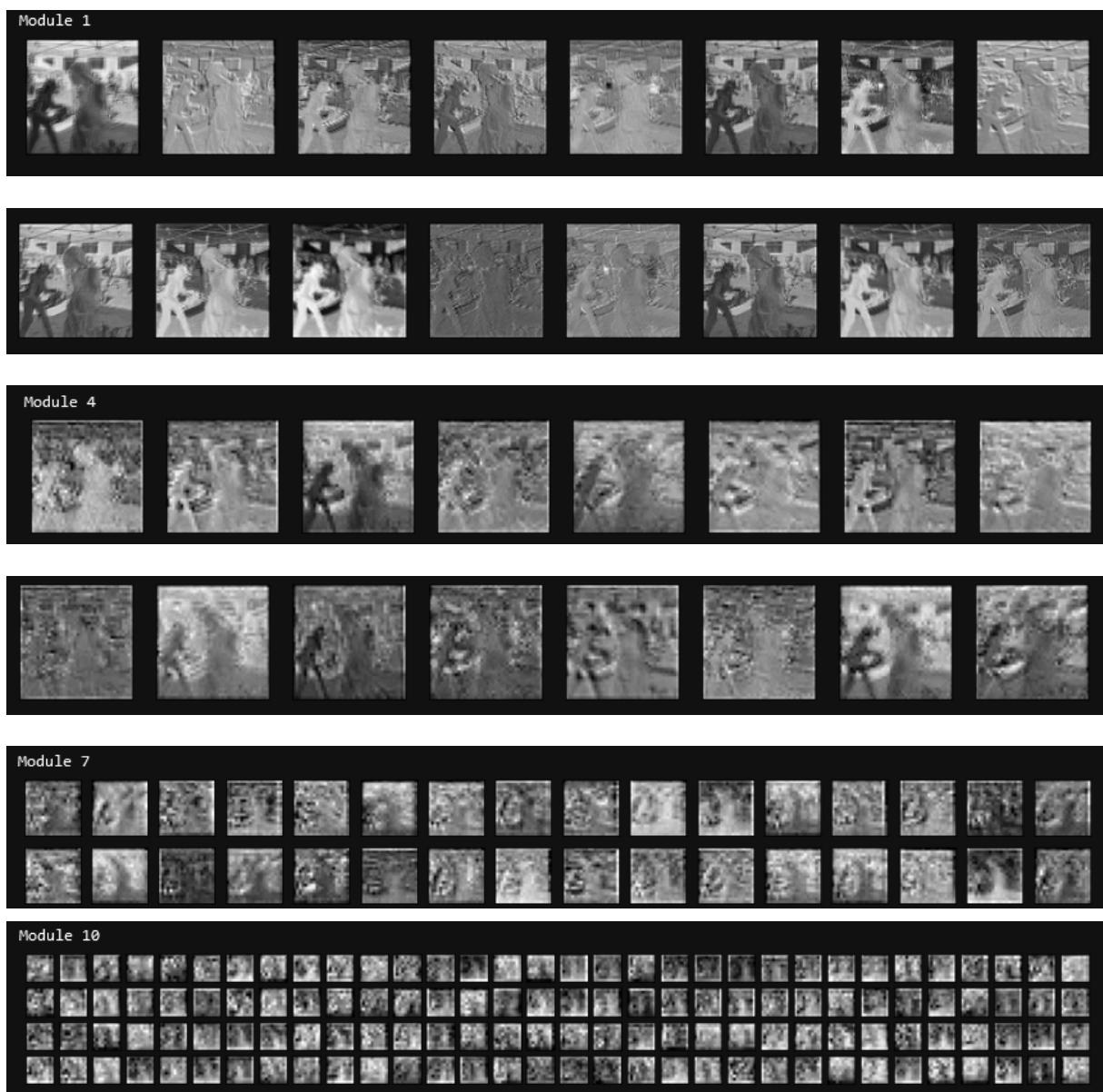
6.4.1. Performance



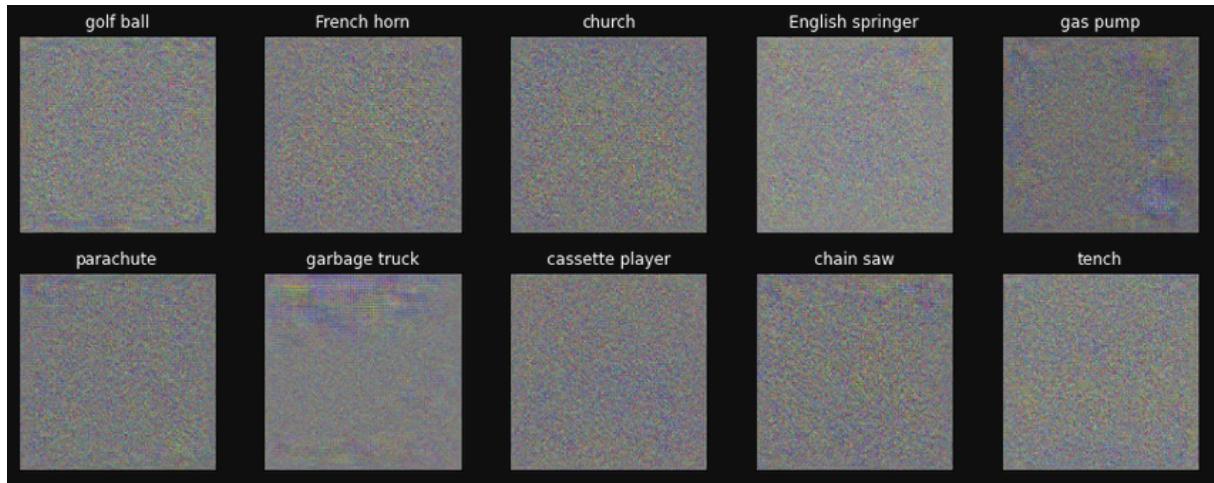
#### 6.4.2. Visualize filters



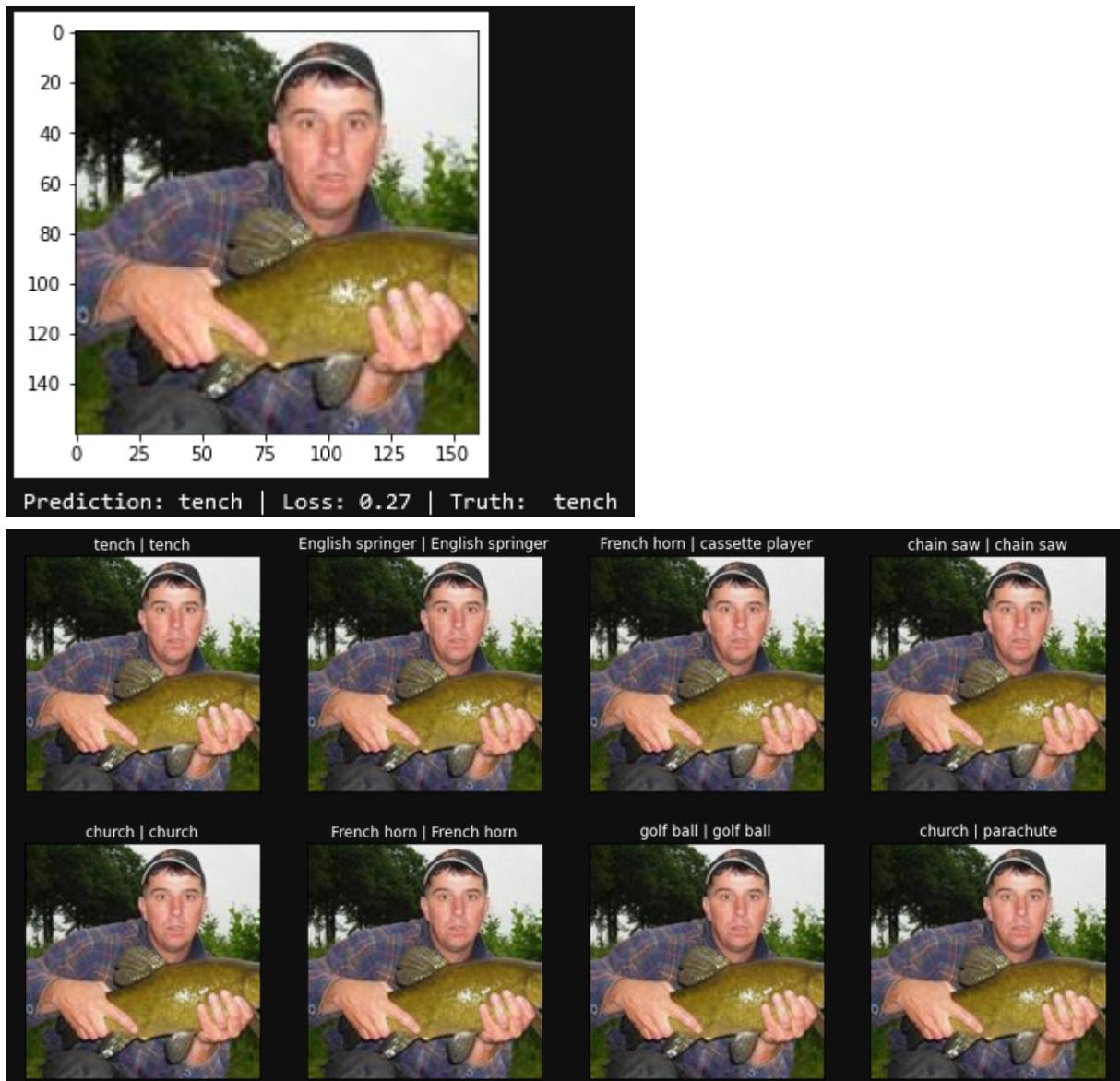
#### 6.4.3. Visualize feature map



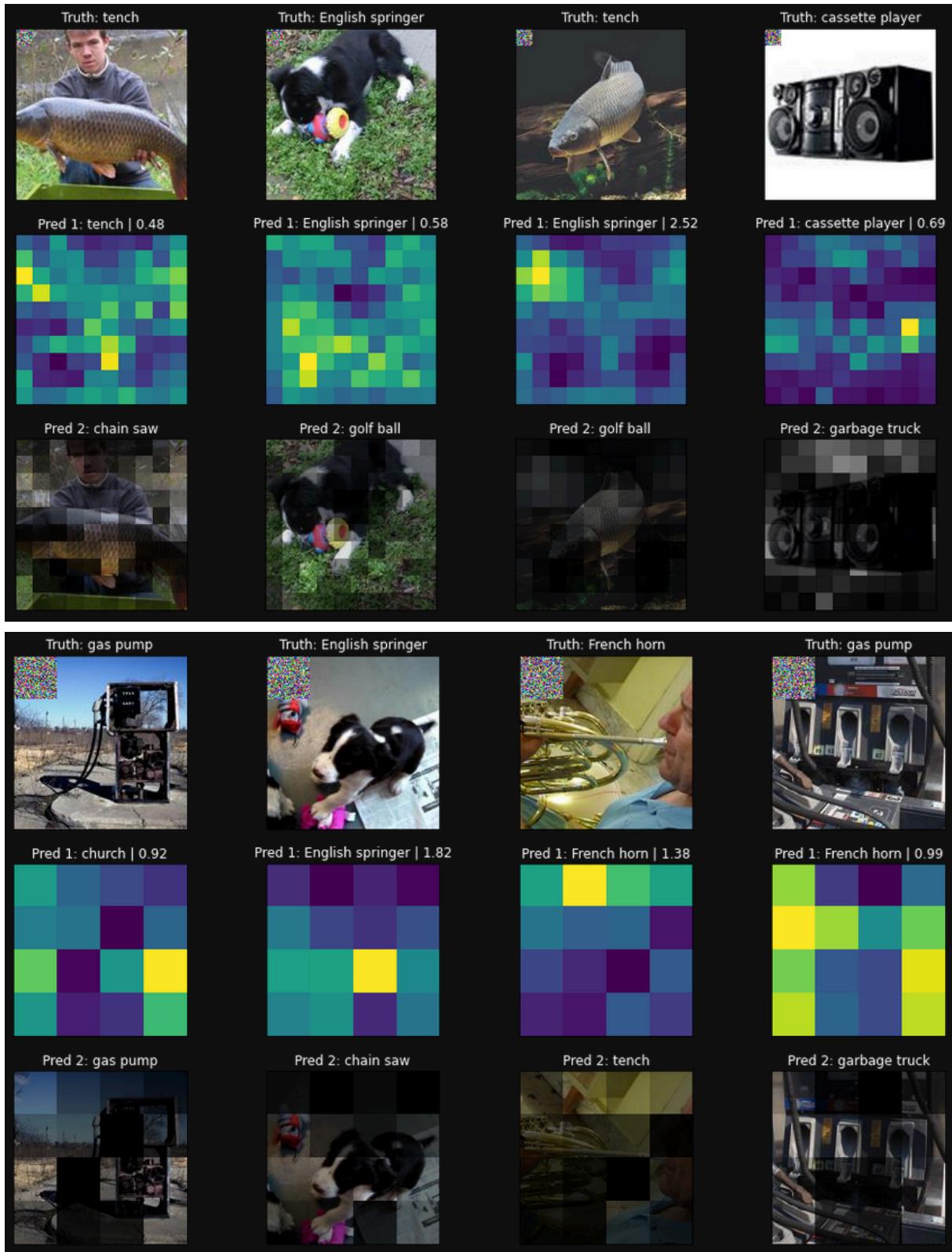
#### 6.4.4. Visualize backward signals



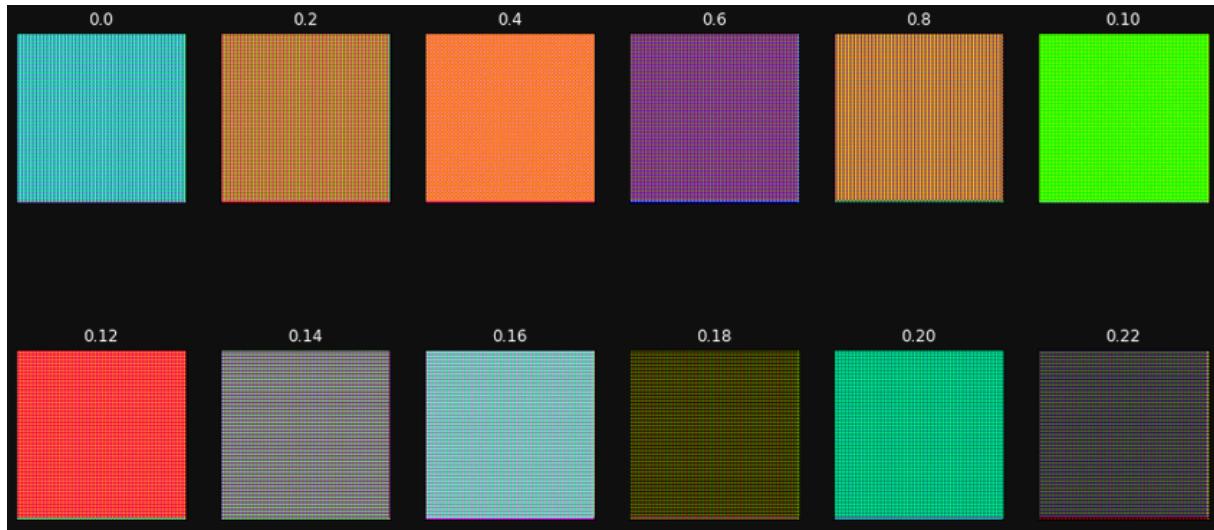
#### 6.4.5. Adversarial attack



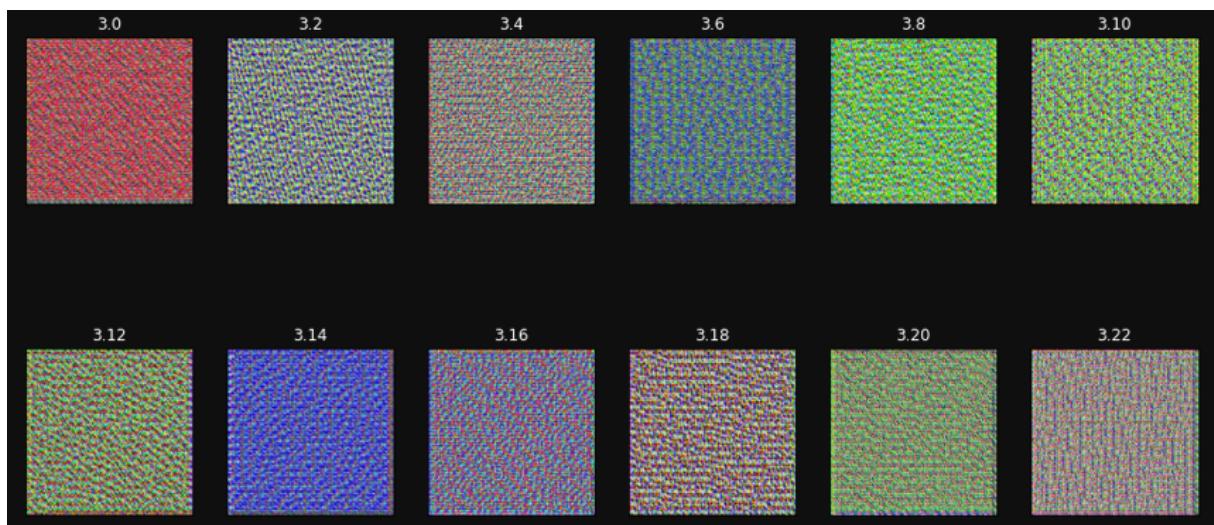
#### 6.4.6. Occlusion test



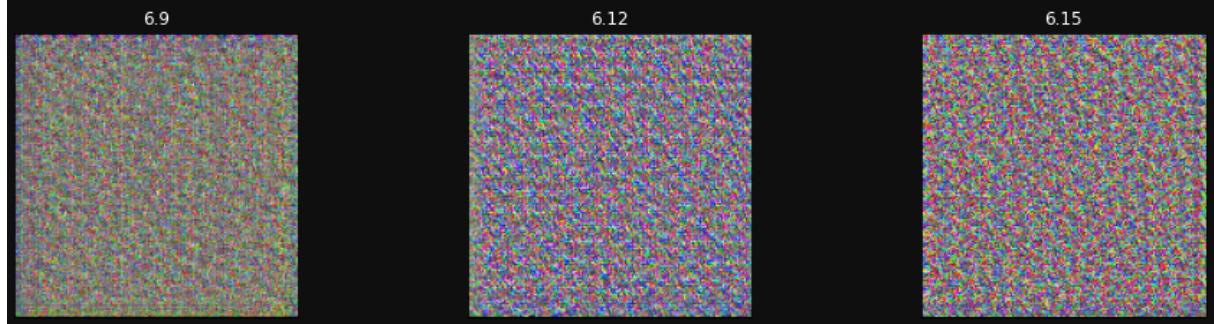
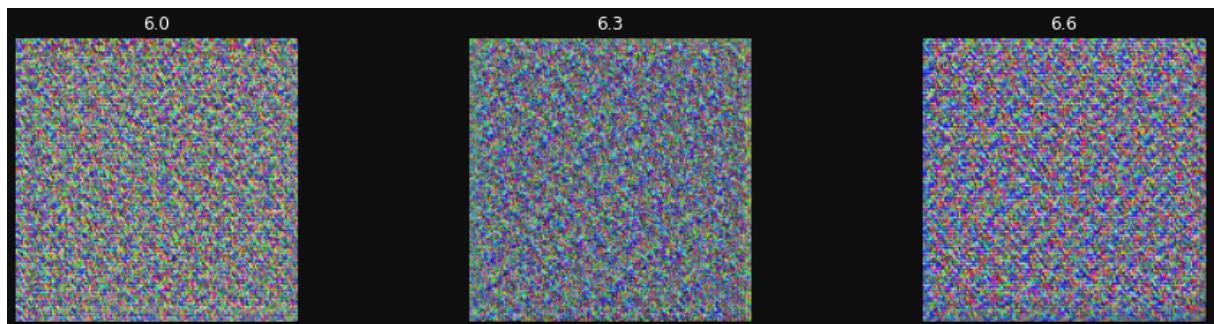
#### 6.4.7. Feature visualization - Filters



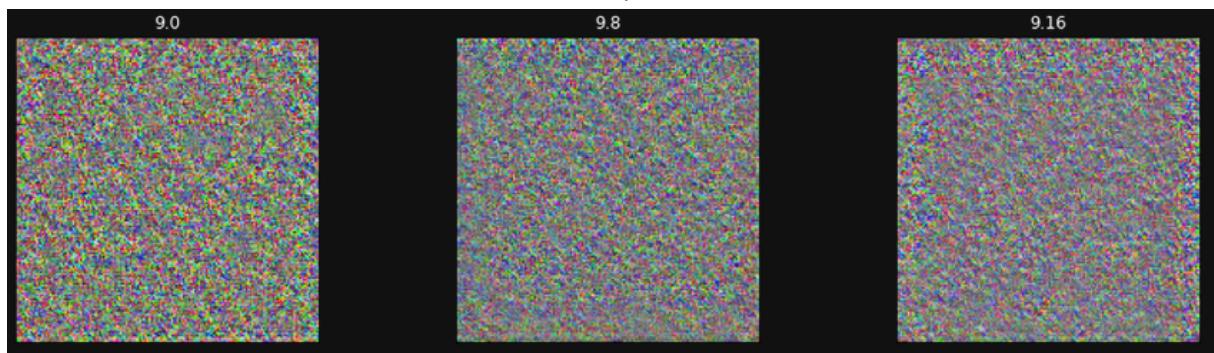
Conv layer 1



Conv layer 2



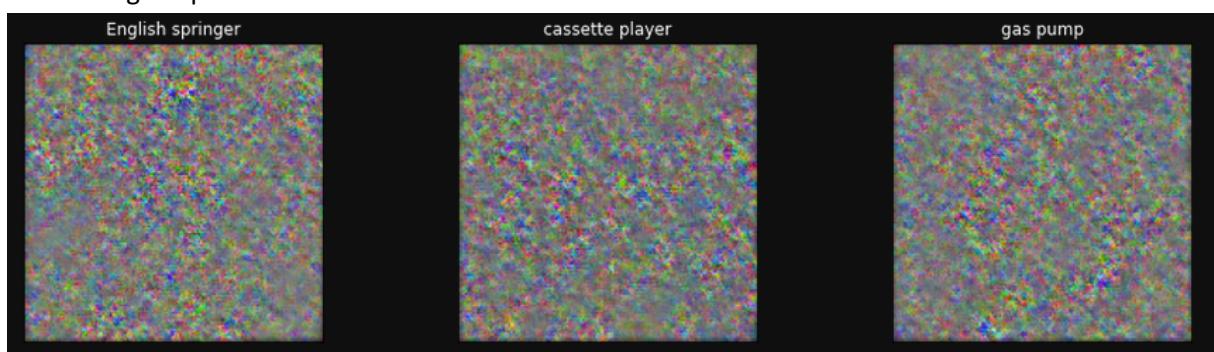
Conv layer 3



Conv layer 4

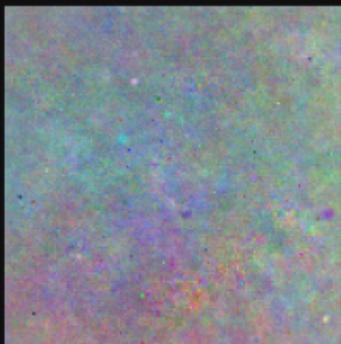
#### 6.4.8. Feature visualization - Class

##### 6.4.8.1. Logits optimization

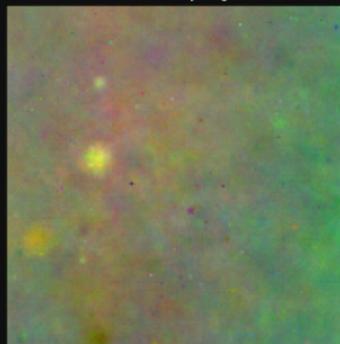


##### 6.4.8.2. Softmax probability optimization

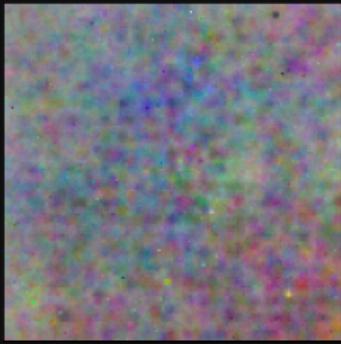
chain saw



cassette player



golf ball



garbage truck

