DEEP LEARNING

Par Kévin Bouchard Ph.D.
Professeur titulaire en intelligence artificielle et apprentissage automatique
Laboratoire d'Intelligence Ambiante pour la reconnaissance d'activités (LIARA)
Directeur de l'Espace innovation en technologies numériques Hydro-Québec
Président du Regroupement québécois des maladies orphelines (RQMO)
Université du Québec à Chicoutimi
www.Kevin-Bouchard.ca Kevin_Bouchard@uqac.ca

CONTENU DE LA LEÇON #3

Vous apprendrez:

- Comment on passe d'un simple neurone à un réseau profond
- Comment on entraîne un réseau profond sans boucle
- L'implémentation concrète et celles assistée par les librairies TF et PyTorch

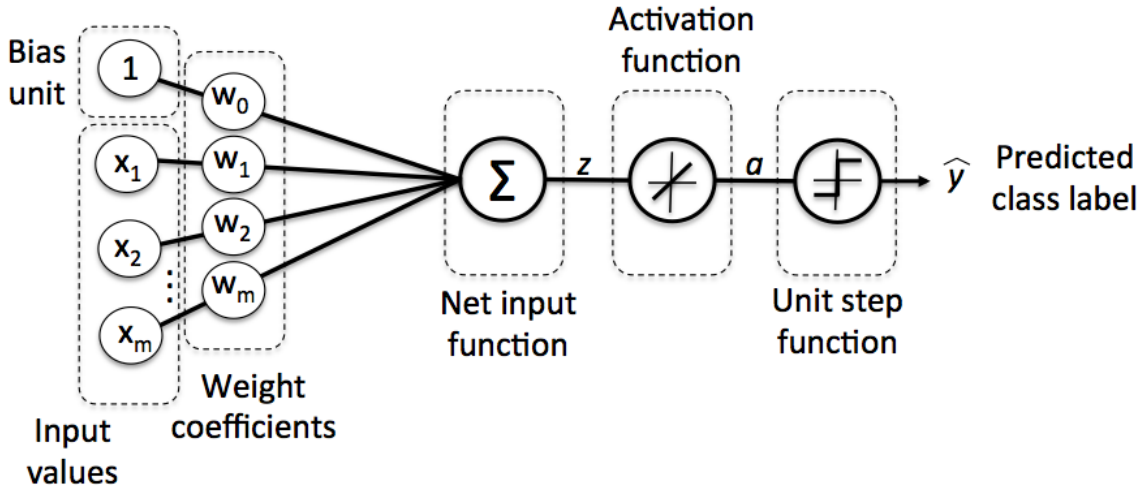
Contenu spécifique:

- Multi-Layer Perceptron
- Propagation avant
- Fonction de coûts
- Rétropropagation
- Exemples de code

INTRODUCTION

- Le deep learning ou l'apprentissage profond est la tendance la plus populaire en apprentissage automatique
- Il s'agit d'un ensemble d'algorithmes développés pour l'entraînement de réseau de neurones artificiels à **plusieurs couches**
- C'est en 1986 que l'intérêt des NN reprend avec la découverte de l'algorithme **backpropagation** qui permet l'entraînement de réseaux multicouches plus facilement
- **State-of-the-art** pour les problèmes exploitant des données complexes (images, voix, textes)
  - Nous verrons plus précisément les propriétés

RETOUR SUR LES NN SIMPLES



- À chaque epoch, on mettait à jour les poids: $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \quad \Delta\mathbf{w} = -\eta\nabla J(\mathbf{w})$
- On utilisait la descente du gradient ou SGD (version itérative)
- La fonction objective à optimiser était la *Sum of Squared Errors* (SSE) ou encore *Log-Likelihood* (NLLL)
  - Selon si Adaline ou Logistic

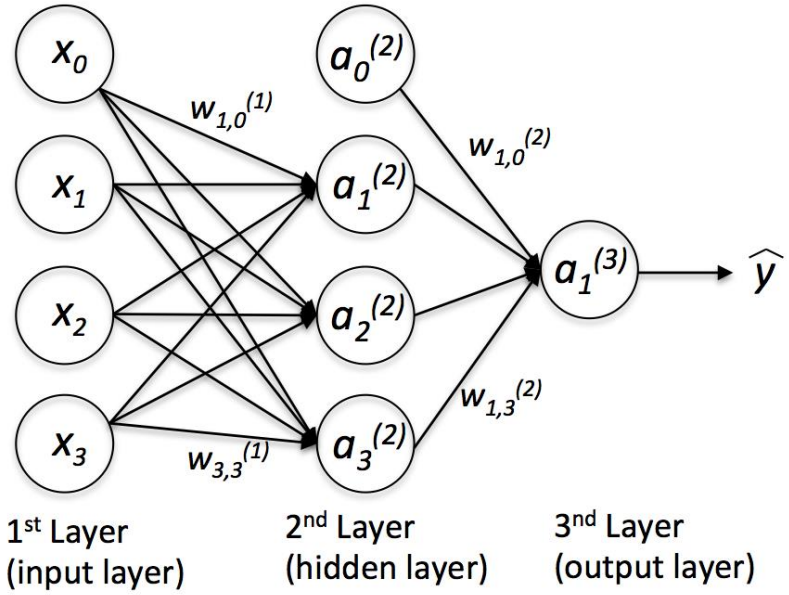
APPRENTISSAGE PAR ENSEMBLE?

- L'apprentissage par ensemble (ensemble learning) repose sur l'idée de combiner plusieurs modèles pour améliorer les performances de prédiction par rapport à un seul modèle
  - Avec le Bagging on réduire la variance et diminue les risques de surapprentissage (e.g. Random Forest)
  - Avec le Boosting on augmente la robustesse en compensant les erreurs individuelles des modèles (e.g. Adaboost, Gradient Boosting)
  - Avec le Stacking, on peut exploiter les forces et les faiblesses de différents modèles en apprenant automatiquement comment « voter »
  - Toutes les techniques tendent à améliorer la justesse (accuracy)!
- En deep learning, nous tentons de tirer profit d'un peu toutes ces techniques!

MULTIPLES COUCHES

- Nos NN sont simple couche, même s'il y a une couche d'entrées et une de sorties à cause du lien unique qui les lie
- On peut ajouter une couche cachée de neurones pour former un réseau de neurones multicouche à propagation avant (multi-layer feed forward NN)

- Les neurones de la couche cachée sont complètement connectés à la couche d'entrées et à la couche de sortie
- Si + d'un niveau caché, alors besoin des techniques du **deep learning**



PERCEPTRON MULTICOUCHE

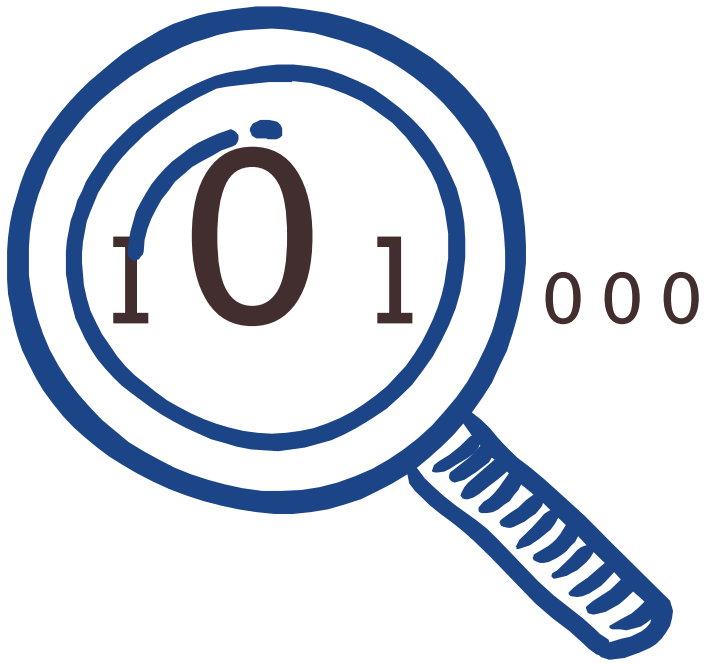
- Nous notons maintenant l'unité d'activation i de la couche ℓ: $a_i^{(l)}$
- L'activation des unités à la couche d'entrée est:

$$\mathbf{a}^{(i)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

- Chaque unité k de la couche ℓ est connecté à l'aide d'un poids à chaque unité j du niveau ℓ +1: $w_{j,k}^{(l)}$
- **Attention!** Il ne faut pas confondre avec l'échantillon i dans $x_m^{(i)}$

MULTICLASSE NATIVEMENT?

- [0 1 0 1 0 0 0 1 0]



```python
import torch
import torch.nn.functional as F

labels = torch.tensor([0, 1, 2, 1, 0])
num_classes = 3

one_hot_labels = F.one_hot(labels, num_classes=num_classes)
```

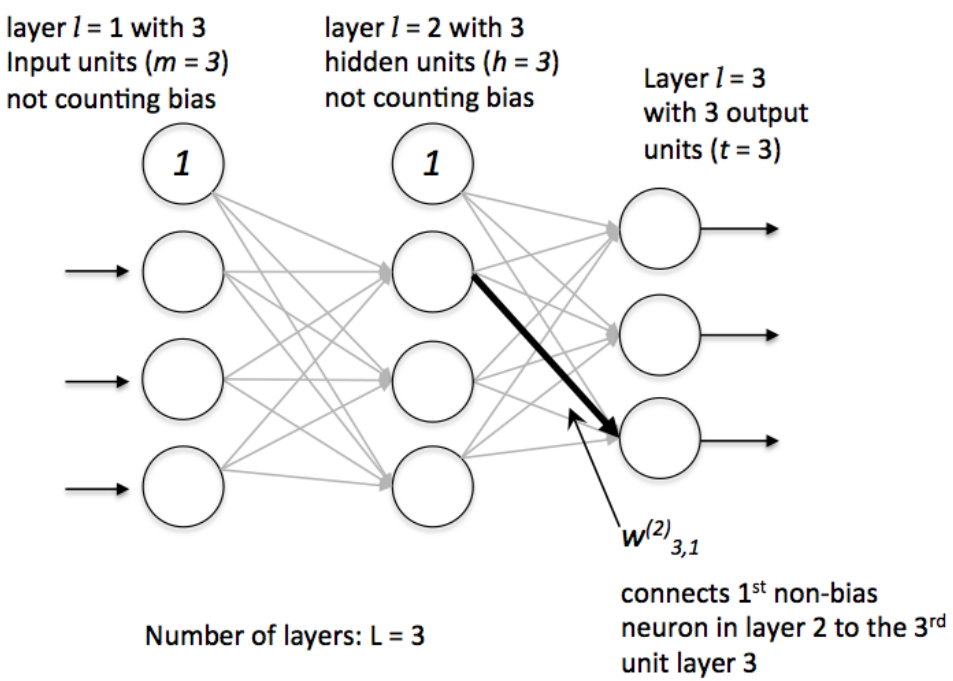
```
Étiquettes d'origine :
tensor([0, 1, 2, 1, 0])

Représentation One-Hot :
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1],
        [0, 1, 0],
        [1, 0, 0]])
```

PERCEPTRON MULTICOUCHE

- Une unité dans la couche de sortie permet de faire de la classification binaire
- À l'aide de la représentation de classe par vecteur **one-hot**, on peut passer aux problèmes multiclasses

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Contrairement à la représentation de classe en entier, les one-hot ne causent pas de problèmes avec les algorithmes basés sur des distances



PERCEPTRON MULTICOUCHE

- Pourquoi $w_{j,k}^{(l)}$ plutôt que $w_{k,j}^{(l)}$? Simplement une question mathématique!

- Matrice $W^{(l)} \in \mathbb{R}^{j \times [k+1]}$ pour représenter les poids de la couche ℓ
  - j est le nombre d'unités à la couche ℓ +1
  - k le nombre d'unités à la couche ℓ
- Donc, un réseau à L couches a L-1 matrices de poids W

PROPAGATION AVANT

- La propagation avant permet de calculer la ou les sorties du réseau multicouche
- Puisque notre réseau est complètement connecté, nous calculons l'activation d'une unité $a_1^{(2)}$ de cette façon:

$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \cdots + a_m^{(1)} w_{1,m}^{(1)}$$

$$a_1^{(2)} = \phi(z_1^{(2)})$$

- $z_1^{(2)}$ est l'entrée nette et $\phi(\cdot)$ la fonction d'activation
  - Celle-ci doit être dérivable en tout point (gradient)
  - Non linéaire pour les problèmes complexes (e.g.:sigmoid)

$$\phi(z) = \frac{1}{1+e^{-z}}$$

PROPAGATION AVANT

- Chaque couche sert d'entrée à la couche suivante sans qu'il n'y ait de boucle
  - Ce n'est pas le cas dans un *recurrent neural network*, où au contraire il y a une boucle!
  - Nous verrons plus tard quelques modèles avec des boucles

- **Attention!** Même si on nomme ce modèle *multi-layer perceptron*, les couches sont composées d'unités **sigmoids**, non pas de perceptrons!
  - De plus, les valeurs retournées par les unités sont continues (0..1)
  - Nous n'utiliserons plus le fameux quantizer en deep learning

PROPAGATION AVANT

- Question de simplicité, nous utiliserons une notation plus compacte pour l'activation:

$$z^{(2)} = W^{(1)} a^{(1)}$$
$$a^{(2)} = \phi(z^{(2)})$$

  - $a^{(1)}$ est le vecteur de features de taille m+1 correspondant à l'échantillon $x^{(i)}$ plus le biais
  - m est le nombre d'entrées dans l'unité
  - $W^{(1)}$ est la matrice de taille $h \times [m+1]$ avec h représentant le nombre d'unités cachées
  - $z^{(2)}$ devient le vecteur d'entrées nettes $h \times 1$ pour l'activation

PROPAGATION AVANT

- De plus, on peut généraliser aux n échantillons:

$$Z^{(2)} = W^{(1)}\left[A^{(1)}\right]^T$$

  - $A^{(1)}$ est la matrice $n \times [m+1]$
  - $Z^{(2)}$ est la matrice $h \times n$

- Avec notre matrice $Z^{(2)}$, nous pouvons calculer la matrice $h \times n$ d'activation du niveau suivant: $A^{(2)} = \phi\left(Z^{(2)}\right)$

PROPAGATION AVANT

- La matrice d'entrées nettes $t \times n$ de la couche suivante (output) devient:

$$Z^{(3)} = W^{(2)} A^{(2)}$$

  - t est le nombre d'unités de la couche de sorties
  - $W^{(2)}$ est la matrice de poids $t \times h$

- Finalement, dans notre exemple, les sorties sont trouvées grâce à l'application de la fonction d'activation sur $Z^{(3)}$

$$A^{(3)} = \phi\left(Z^{(3)}\right)$$

  - $A^{(3)}$ est la matrice de réels de taille $t \times n$, où t est le nombre de classes et n le nombre d'échantillons

PROPAGATION AVANT AVEC CHIFFRES!

$$a^{(1)} = \begin{bmatrix}1\\2\\3\end{bmatrix}, W^{(1)} = \begin{bmatrix}0.5 & 1 & -0.5\\-1 & 2 & 1.5\end{bmatrix}\begin{bmatrix}0.25\\0.5\end{bmatrix}, W^{(2)} = \begin{bmatrix}0 & 0.1\\-1 & 1\\0.1 & 0\end{bmatrix}\begin{bmatrix}1\\-2\\-1\end{bmatrix}$$

biais

biais

$$z^{(2)} = W^{(1)}a^{(1)} = \begin{bmatrix}0.5 & 1 & -0.5\\-1 & 2 & 1.5\end{bmatrix}\begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}0.25\\0.5\end{bmatrix} =$$

$$\begin{bmatrix}0.5*1+1*2-0.5*3\\-1*1+2*2+1.5*3\end{bmatrix} + \begin{bmatrix}0.25\\0.5\end{bmatrix} = \begin{bmatrix}1\\7.5\end{bmatrix} + \begin{bmatrix}0.25\\0.5\end{bmatrix} = \begin{bmatrix}1.25\\8\end{bmatrix}$$

$$a^{(2)} = \phi\left(z^{(2)}\right) = \begin{bmatrix}\frac{1}{1+e^{-1.25}}\\ \frac{1}{1+e^{-8}}\end{bmatrix} \approx \begin{bmatrix}0.7773\\0.9997\end{bmatrix}$$

PROPAGATION AVANT AVEC CHIFFRES!

$$a^{(1)} = \begin{bmatrix}1\\2\\3\end{bmatrix}, W^{(1)} = \begin{bmatrix}0.5 & 1 & -0.5\\-1 & 2 & 1.5\end{bmatrix}\begin{bmatrix}0.25\\0.5\end{bmatrix}, W^{(2)} = \begin{bmatrix}0 & 0.1\\-1 & 1\\0.1 & 0\end{bmatrix}\begin{bmatrix}1\\-2\\-1\end{bmatrix}$$

biais

biais

$$z^{(3)} = W^{(2)}a^{(2)} = \begin{bmatrix}0 & 0.1\\-1 & 1\\0.1 & 0\end{bmatrix}\begin{bmatrix}0.7773\\0.9997\end{bmatrix} + \begin{bmatrix}1\\-2\\-1\end{bmatrix} =$$

$$\begin{bmatrix}0 * 0.7773 + 0.1 * 0.9997\\-1 * 0.7773 + 0.9997\\0.1 * 0.7773 + 0 * 0.9997\end{bmatrix} + \begin{bmatrix}1\\-2\\-1\end{bmatrix} = \begin{bmatrix}0.09997\\0.2224\\0.07773\end{bmatrix} + \begin{bmatrix}1\\-2\\-1\end{bmatrix} = \begin{bmatrix}1.09997\\-2.2224\\-1.07773\end{bmatrix}$$

$$a^{(3)} = \phi\left(z^{(3)}\right) = \begin{bmatrix}\frac{1}{1+e^{-1.09997}}\\\frac{1}{1+e^{2.2224}}\\\frac{1}{1+e^{1.07773}}\end{bmatrix} \approx \begin{bmatrix}0.7503\\0.0978\\0.2539\end{bmatrix}$$

EXEMPLE APPLIQUÉ

CLASSIFICATION DE CHIFFRES

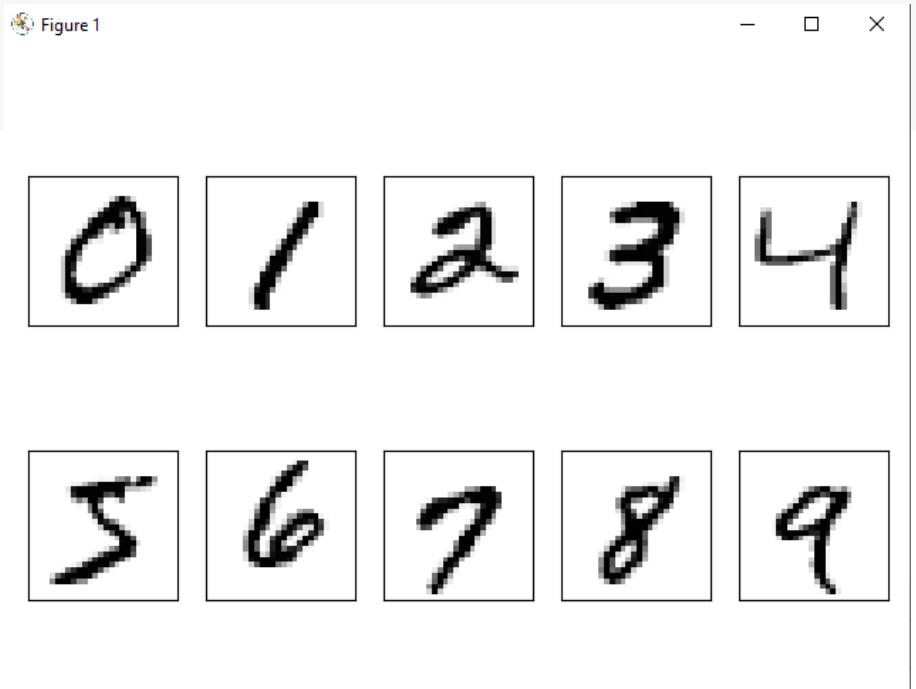
- Nous utiliserons le dataset très connu MNIST qui contient:
  - 60 000 images d'entraînement
  - 10 000 images de tests
  - Les images sont les chiffres 0 à 9 écris à la main
- http://yann.lecun.com/exdb/mnist/

QUELQUES RÉSULTATS SUR LA PAGE:

CLASSIFIER	PREPROCESSING	TEST ERROR RATE (%)
Linear Classifiers		
linear classifier (1-layer NN)	none	12.0
linear classifier (1-layer NN)	deskewing	8.4
pairwise linear classifier	deskewing	7.6
K-Nearest Neighbors		
K-nearest-neighbors, Euclidean (L2)	none	5.0
K-nearest-neighbors, Euclidean (L2)	none	3.09
K-nearest-neighbors, L3	none	2.83
K-nearest-neighbors, Euclidean (L2)	deskewing	2.4
K-nearest-neighbors, Euclidean (L2)	deskewing, noise removal, blurring	1.80
Boosted Stumps		
boosted stumps	none	7.7
products of boosted stumps (3 terms)	none	1.26
boosted trees (17 leaves)	none	1.53
stumps on Haar features	Haar features	1.02
product of stumps on Haar f.	Haar features	0.87
Non-Linear Classifiers		
40 PCA + quadratic classifier	none	3.3
1000 RBF + linear classifier	none	3.6
SVMs		
SVM, Gaussian Kernel	none	1.4
SVM deg 4 polynomial	deskewing	1.1
Reduced Set SVM deg 5 polynomial	deskewing	1.0
Virtual SVM deg-9 poly [distortions]	none	0.8
Virtual SVM, deg-9 poly, 1-pixel jittered	none	0.68
Virtual SVM, deg-9 poly, 1-pixel jittered	deskewing	0.68
Virtual SVM, deg-9 poly, 2-pixel jittered	deskewing	0.56
Neural Nets		
2-layer NN, 300 hidden units, mean square error	none	4.7
2-layer NN, 300 HU, MSE, [distortions]	none	3.6
2-layer NN, 300 HU	deskewing	1.6
2-layer NN, 800 HU, Cross-Entropy Loss	none	1.6
2-layer NN, 800 HU, cross-entropy [affine distortions]	none	1.1
2-layer NN, 800 HU, MSE [elastic distortions]	none	0.9
Convolutional nets		
Convolutional net LeNet-1	subsampling to 16x16 pixels	1.7
Convolutional net LeNet-4	none	1.1
Convolutional net LeNet-4 with K-NN instead of last layer	none	1.1
Convolutional net LeNet-4 with local learning instead of last layer	none	1.1
large conv. net, unsup pretraining [no distortions]	none	0.53
large/deep conv. net, 1-20-40-60-80-100-120-120-10 [elastic distortions]	none	0.35
committee of 7 conv. net, 1-20-P-40-P-150-10 [elastic distortions]	width normalization	0.27 +-0.02
committee of 35 conv. net, 1-20-P-40-P-150-10 [elastic	width normalization	0.23

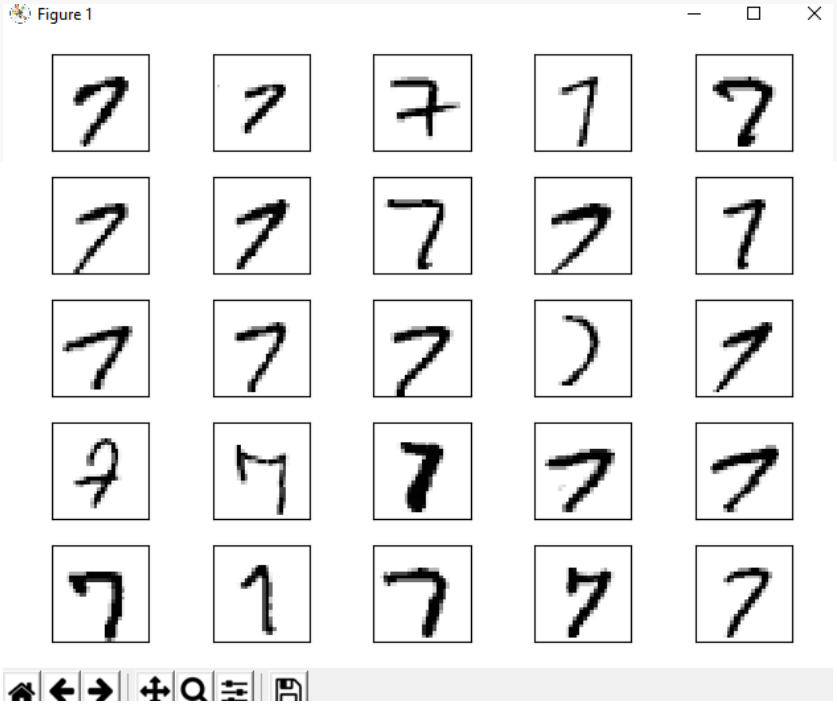
VISUALISATION

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in (10):
    img = X_train[y_train == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout() plt.show()
```



VISUALISATION (SUITE)

```
fig, ax = plt.subplots(nrows=5, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in (25):
    img = X_train[y_train == 7][i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout() plt.show()
```



IMPLÉMENTATION ET TEST

```
nn = NeuralNetMLP(
      n_output=10,
      n_features=X_train.shape[1],
      n_hidden=50,
      l2=0.1, l1=0.0,
      epochs=1000,
      eta=0.001,
      alpha=0.001,
      decrease_const=0.00001,
      minibatches=50,
      shuffle=True, random_state=1)
```

- L2 réduit le surapprentissage
- 784 unités d'entrées
- 50 unités cachées
- 10 unités de sorties (classes)
- Alpha ajoute un momentum du gradient de l'epoch précédent pour accélérer l'apprentissage

$$\Delta w_t = \eta \nabla J(w_t) + \alpha \Delta w_{t-1}$$

- Decrease (d) permet de réduire le taux d'apprentissage au fil du temps

$$\eta/(1 + t \times d)$$

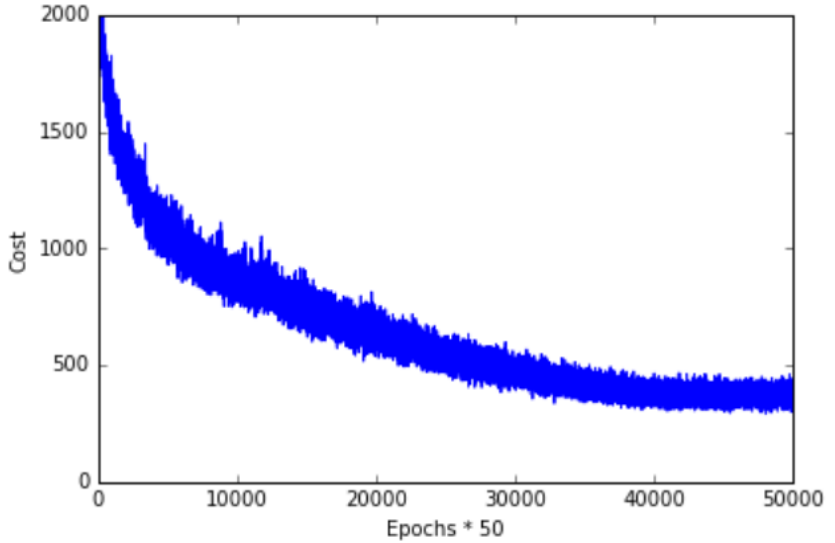
IMPLÉMENTATION ET TEST

- Entraînement du modèle:

```
nn.fit(X_train, y_train, print_progress=False)
```

- Graphique des coûts (cost_) pour chaque minibatchs (50*1000epochs):

```
import matplotlib.pyplot as plt
plt.plot(range(len(nn.cost_)), nn.cost_)
plt.ylim([0, 2000])
plt.ylabel('Cost')
plt.xlabel('Epochs * 50')
plt.tight_layout()
plt.show()
```



Voir https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

ÉVALUATION

- Évaluons le modèle avec l'ensemble de test:

```
y_test_pred = nn.predict(X_test)
acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (acc * 100))

Test accuracy: 95.62%
```

- Le modèle pourrait être amélioré à l'aide de l'optimisation des hyperparamètres
  - Celui-ci en contient plusieurs: learning rate, nombre d'unités cachées, alpha, decrease_const et régularisation L1 & L2
  - Cette partie peut s'avérer assez complexe dans certains problèmes d'apprentissage

IMPLÉMENTATION EN MINIBATCHS

- L'utilisation des minibatchs pour le calcul de nos gradients est un type particulier de *stochastic gradient descent*
  - Plutôt que de calculer sur un échantillon, nous le faisons sur k
  - $1 < k < n$

- Plus rapide que gradient descent (k=n)
- Mais plus efficace en implémentation que SGD (k=1)
  - Implémentation des techniques de calculs vectoriels!
  - Beaucoup plus efficace!

- C'est un peu comme un sondage avant des élections…

AVEC TF/KERAS

```
import tensorflow_datasets as tfds

(ds_train, ds_test), ds_info = tfds.load(
    'mnist',
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    with_info=True,
)


def normalize_img(image, label):  return tf.cast(image, tf.float32) / 255., label

ds_train = ds_train.map(normalize_img)
ds_train = ds_train.shuffle(ds_info.splits['train'].num_examples)
ds_train = ds_train.batch(128)

ds_test = ds_test.map(normalize_img)
ds_test = ds_test.batch(128)


model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
  tf.keras.layers.Dense(128,activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

```
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=tf.keras.optimizers.Adam(0.001),
    metrics=['accuracy'],
)

model.fit(
    ds_train,
    epochs=6,
    validation_data=ds_test,
)
```

EXÉCUTION DE L'EXEMPLE

```
469/469 [==============================] - 1s 3ms/step - loss: 0.3595 - accuracy: 0.9010 - val_loss: 0.1905 - val_accuracy: 0.9483
Epoch 2/6
469/469 [==============================] - 1s 2ms/step - loss: 0.1612 - accuracy: 0.9543 - val_loss: 0.1351 - val_accuracy: 0.9593
Epoch 3/6
469/469 [==============================] - 1s 2ms/step - loss: 0.1143 - accuracy: 0.9678 - val_loss: 0.1051 - val_accuracy: 0.9689
Epoch 4/6
469/469 [==============================] - 1s 2ms/step - loss: 0.0878 - accuracy: 0.9750 - val_loss: 0.0981 - val_accuracy: 0.9708
Epoch 5/6
469/469 [==============================] - 1s 2ms/step - loss: 0.0707 - accuracy: 0.9797 - val_loss: 0.0846 - val_accuracy: 0.9748
Epoch 6/6
469/469 [==============================] - 1s 2ms/step - loss: 0.0583 - accuracy: 0.9830 - val_loss: 0.0801 - val_accuracy: 0.9755
```

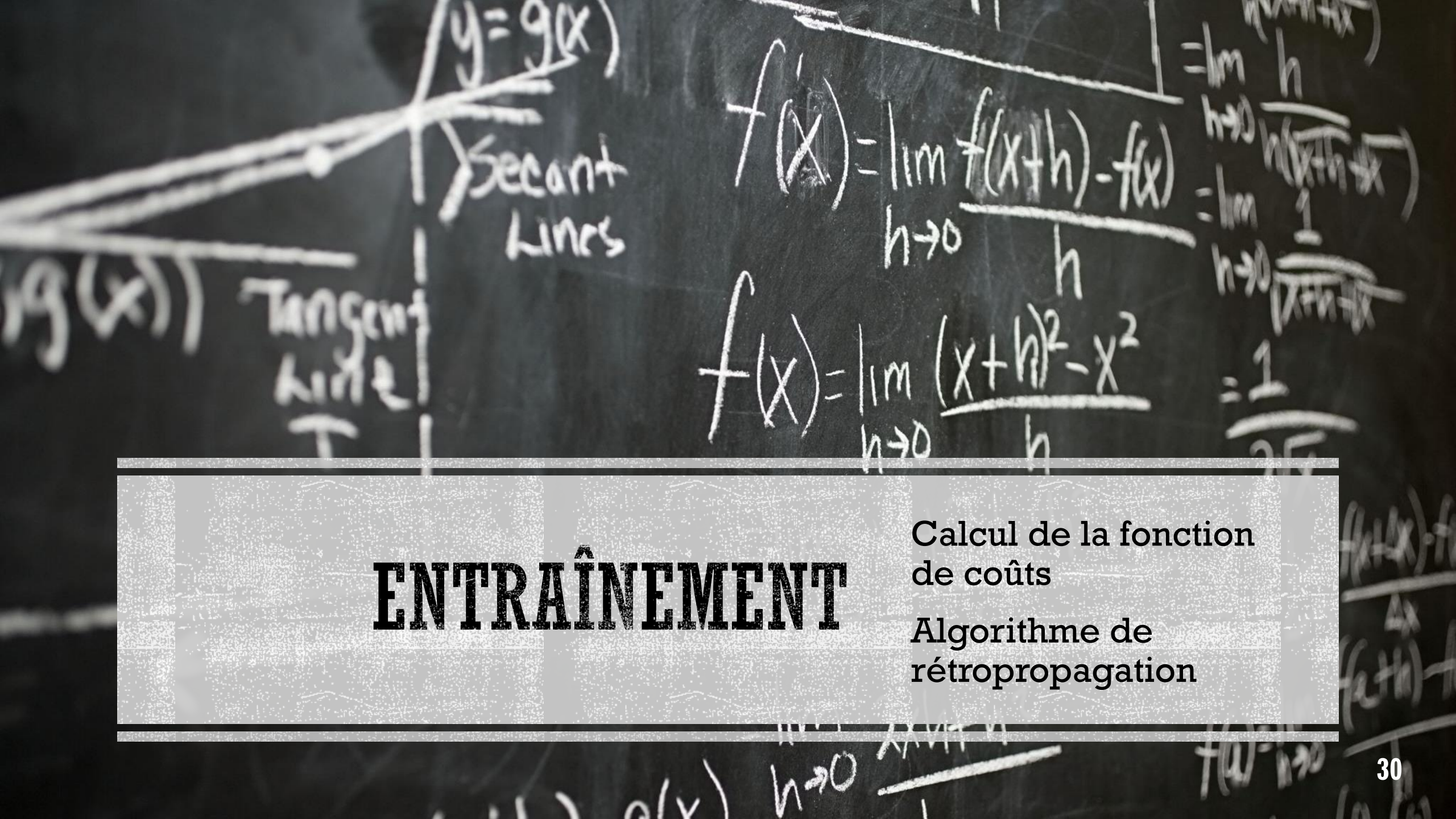
- On peut voir que c'est beaucoup plus rapide avec Tensorflow que notre code
- Les résultats sont aussi meilleurs, même si nous n'avons rien optimisé
- En quelques lignes, nous avons un modèle qui dépasse la majorité des méthodes d'apprentissage automatique classique!

AVEC PYTORCH

```
Epoch [1/5], Loss: 0.1187
Epoch [2/5], Loss: 0.1095
Epoch [3/5], Loss: 0.0239
Epoch [4/5], Loss: 0.1840
Epoch [5/5], Loss: 0.1625
Précision sur l'ensemble de test : 96.30%
```

```python
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize( mean: (0.5,), std: (0.5,))
])

# Charger l'ensemble d'entraînement et de test
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)


class MLP(nn.Module):  2 usages
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Aplatir les images 28x28 en vecteurs
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

```python
model = MLP(input_size=28*28, hidden_size=128, num_classes=10)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    model.train()
    for batch_idx, (images, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch [{epoch + 1}/{5}], Loss: {loss.item():.4f}")
```



ENTRAÎNEMENT

Calcul de la fonction de coûts

Algorithme de rétropropagation

FONCTION DE COÛTS

- La fonction de coût ici est la même que pour logistic regression:

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

  - $a^{(i)}$ est l'activation sigmoid de l'unité d'une couche: $\phi(z^{(i)})$
  - Pour un échantillon i

- En ajoutant la régularisation L2 (pour réduire le surapprentissage) on obtient:

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right] + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

FONCTION DE COÛTS

- Puisque nous voulons faire de la classification multi classes, le vecteur de sortie est de taille $t \times 1$ à comparer avec la cible dans le vecteur one hot (lui aussi de taille *t)*
- E.g. l'activation à la couche de sortie (3) et la comparaison avec la classe cible #2 pourrait ressembler à ceci:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

FONCTION DE COÛTS

- Il faut généraliser notre fonction de coûts à toutes les unités *j*:
  - Ici, (i) représente l'échantillon d'entraînement

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{m} y_j^{(i)} \log\left(\phi\left(z_j^{(i)}\right)\right) + \left(1-y_j^{(i)}\right) \log\left(1-\phi\left(z_j^{(i)}\right)\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

  - L'équation fait un peu peur, mais elle représente simplement les coûts pour toutes les unités du réseau et tous les échantillons de l'ensemble d'entraînement

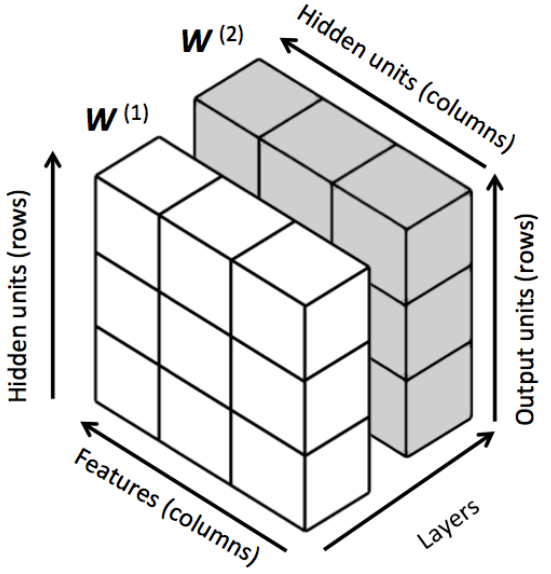
  - De plus, $\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$ est la pénalité L2 (l est la couche!)

FONCTION DE COÛTS

- **Objectif**: minimiser notre fonction de coûts
- Il faut donc calculer la dérivée partielle de la matrice de poids par rapport à tous les poids du réseau:

$$\frac{\partial}{\partial w_{j,i}^{l}} J(\mathbf{W}).$$

  - À noter que W est en fait un ensemble de matrices qui n'ont généralement pas la même dimension
  - Leur taille dépend du nombre d'unités des différents niveaux



```
def _get_cost(self, y_enc, output, w1, w2):
    """Compute cost function.

    Parameters
    ----------
    y_enc : array, shape = (n_labels, n_samples)
        one-hot encoded class labels.
    output : array, shape = [n_output_units, n_samples]
        Activation of the output layer (feedforward)
    w1 : array, shape = [n_hidden_units, n_features]
        Weight matrix for input layer -> hidden layer.
    w2 : array, shape = [n_output_units, n_hidden_units]
        Weight matrix for hidden layer -> output layer.

    Returns
    ---------
    cost : float
        Regularized cost.

    """
    term1 = -y_enc * (np.log(output))
    term2 = (1.0 - y_enc) * np.log(1.0 - output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1, w1, w2)
    L2_term = self._L2_reg(self.l2, w1, w2)
    cost = cost + L1_term + L2_term
    return cost
```

CODE

- L1 tend à éliminer des caractéristiques
- C'est un peu du feature sélection!
- L2 diminue les valeurs des poids, mais sans atteindre 0

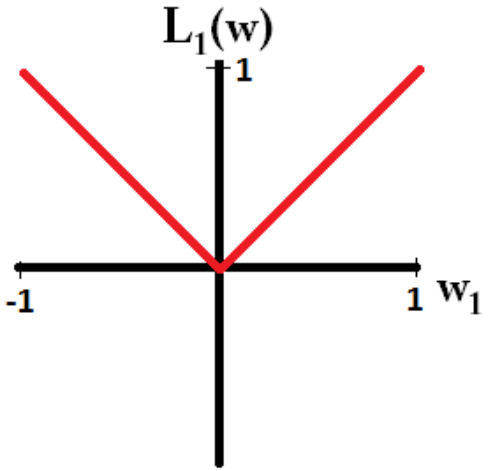
FONCTIONS SIMPLES

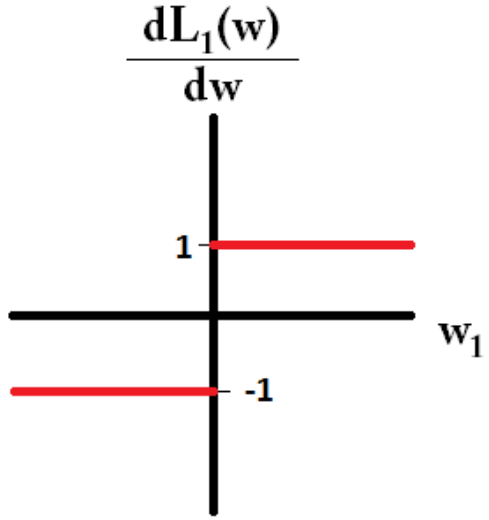
- Ok, mais concrètement, comment L1 peut-elle faire de la sélection de caractéristiques?
- En fait, si on regarde l'implémentation du gradient, nous avons :

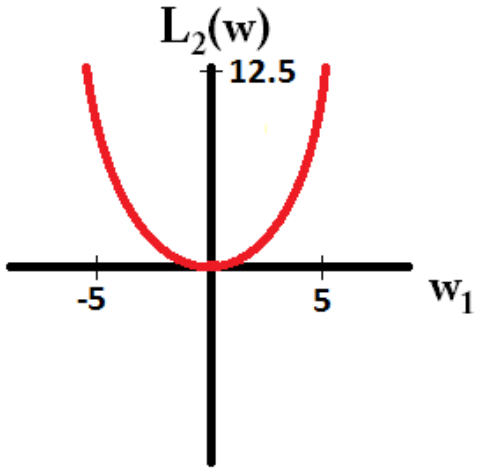
```
# regularize
grad1[:, 1:] += self.l2 * w1[:, 1:]
grad1[:, 1:] += self.l1 * np.sign(w1[:, 1:])
```

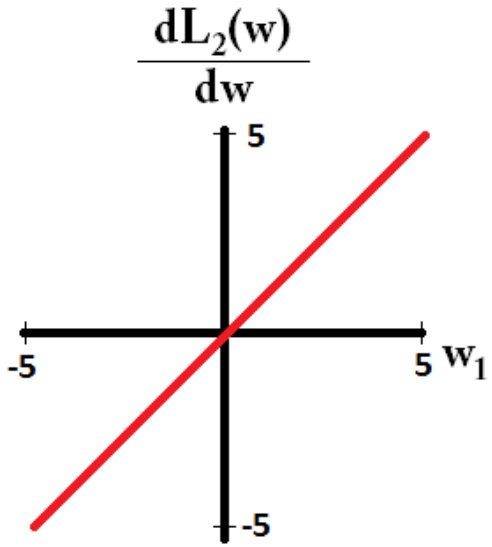
- Soit $\Delta \mathrel{+}= \lambda * W^{(l)}$ pour L2 et $\Delta \mathrel{+}= \begin{cases} \lambda * 1, w > 0 \\ \lambda * -1, w < 0 \end{cases}, \forall w \in W^{(l)}$ pour L1
- C'est qu'en dérivant $W^{(l)} = W^{(l)} - \eta\Delta^{(l)}$, sachant que delta est la dérivée de la fonction de coûts $\frac{\partial J(W)}{\partial W}$ et qu'on y additionne simplement les termes L1 et L2

FONCTIONS SIMPLES









- La conséquence est que pour L1, si une soustraction d'un w a pour effet d'en changer le signe
- Par exemple de +0.3 à -0.1
- Alors on croise le 0 et L1 fait passer le w à 0

BACKPROPAGATION

Permet d'apprendre les poids dans un réseau de neurones efficacement

Nous verrons d'abord intuitivement comment la méthode fonctionne, puis nous décrirons un peu plus formellement

BACKPROPAGATION

- Algorithme efficace (d'un point de vue algorithmique) qui permet de calculer les dérivées d'une fonction de coûts complexe
  - On utilise les dérivées pour apprendre les poids (comme d'habitude)
  - Dans le contexte de nos réseaux multicouches, nous avons beaucoup de poids et travaillons généralement en très haute dimension!

- Nos fonctions de coûts ne sont plus convexes et lisses, elles sont concaves et inégales
  - Cas plus difficile en optimisation!!!

BACKPROPAGATION

- La rétropropagation est utilisée dans la majorité des applications courantes de l'apprentissage profond, même si elle est souvent cachée aux développeurs
- Cependant, voici quelques exceptions si vous souhaitez aller au-delà du cadre de ce cours:
- Réseaux de neurones utilisant des algorithmes évolutionnaires : Au lieu d'utiliser les gradients pour optimiser les poids, ces algorithmes simulent le processus de sélection naturelle
- Hebbian Learning: Un apprentissage non supervisé basé sur la théorie biologique de Hebb
- Echo State Networks (ESNs): Un type de réseau de neurones récurrents où les poids internes du réseau ne sont pas entraînés (pas besoin de rétropropagation)
- Quantum Neural Networks (QNN): Certaines implémentations de réseaux de neurones quantiques n'utilisent pas la rétropropagation, car le processus d'entraînement repose sur des principes de calcul quantique qui ne dépendent pas de la descente de gradient

BACKPROPAGATION

- L'intuition derrière l'algorithme de rétropropagation est de renverser une opération pour réduire les coûts en calculs
  - En avant, il faut successivement multiplier des matrices ensemble
  - En arrière, on part d'un vecteur qu'on multiplie par une matrice et qui résulte en un autre vecteur à multiplier par une autre matrice…

- On se rappelle que pour notre MLP, nous avons appliqué la propagation avant de cette façon:

$$Z^{(2)} = W^{(1)}\left[A^{(1)}\right]^T$$

Entrées nettes de la couche caché

$$A^{(2)} = \phi\left(Z^{(2)}\right)$$

$$Z^{(3)} = W^{(2)}A^{(2)}$$

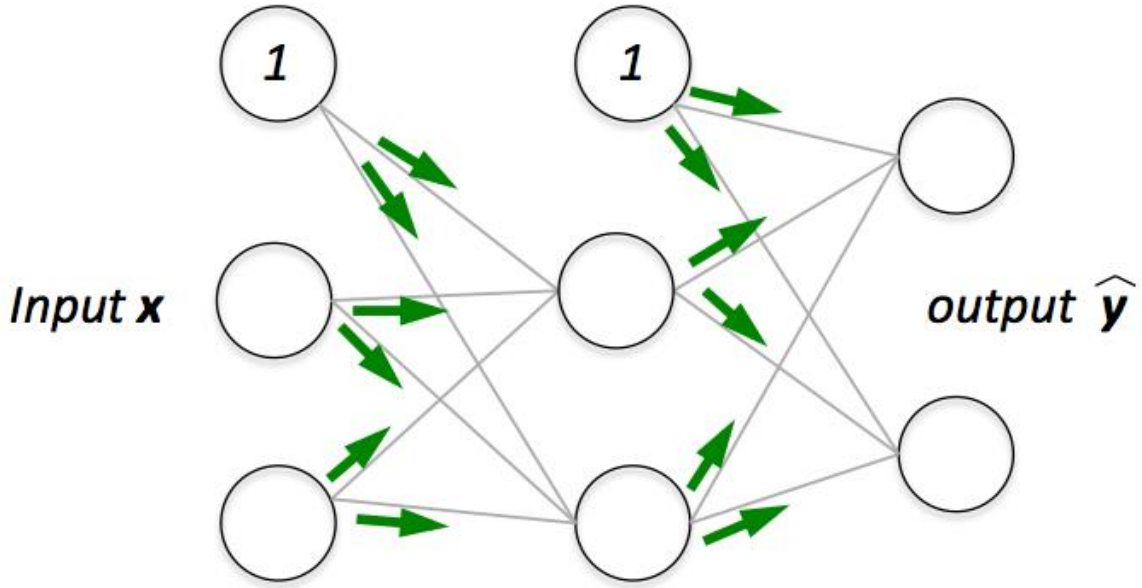
Entrées nettes de la couche de sorties

$$A^{(3)} = \phi\left(Z^{(3)}\right)$$

Appartenance aux classes

BACKPROPAGATION

- Nous propageons par avant les caractéristiques en entrées via les connections du réseau
- De façon plus visuelle, voici ce que nous faisons:

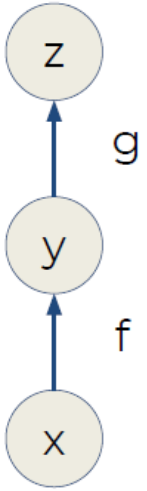


COMPOSITION DE FONCTION

- Si notre sortie est la composition des fonctions $g \circ f(x) = g(f(x))$
- Alors la composition de dérivées est $\big(g \circ f(x)\big)'(x) = g'\big(f(x)\big)f'(x)$
- Bref, si nous avons un graphe du type:

$z = g(y)$

$y = f(x)$



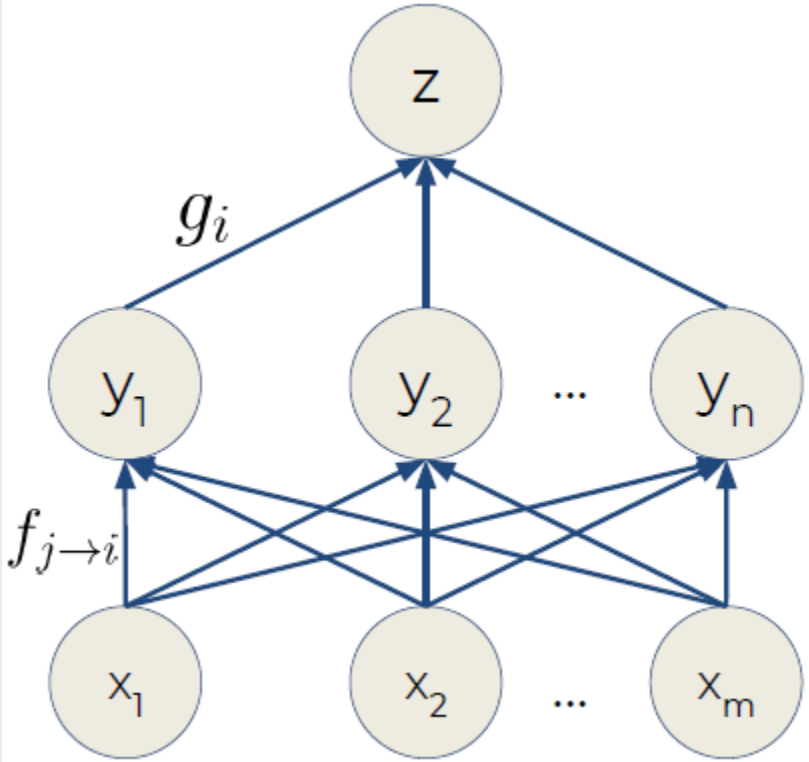
Dans graphe computationnel, les variations des variables sont liées par les dérivées partielles :

$\Delta z \approx g'\big(f(x)\big)\Delta y$ ou encore: $\Delta z \approx g'\big(f(x)\big)\, f'(x)\Delta x$

Le changement dans y est donc $\Delta y \approx f'(x)\Delta x$

La variation ou perturbation de x est Δx

GRAPHE COMPUTATIONNEL



$z = \sum_i g_i(y_i)$

Soit $\Delta z \approx \sum_i g'_i(\sum_j f_{j\to i}(x_j))\,\Delta y_i$

Et… $\Delta z \approx \sum_i g'_i(\sum_j f_{j\to i}(x_j)) \sum_j f'_{j\to i}(x_j)\,\Delta x_j$

$y_i = \sum_j f_{j\to i}(x_j)$

Soit $\Delta y_i \approx \sum_j f'_{j\to i}(x_j)\,\Delta x_j$

$\Delta x_1, \ldots, \Delta x_m$

LES GRADIENTS DE Z

- Grosso modo, on pourrait récrire la variation de la sortie Δz comme le produit scalaire:

$$\Delta z = \langle \nabla z, \Delta x \rangle$$

- Sachant que ∇z est le vecteur de gradients qui capture les sensibilités partielles de z par rapport aux x_j et la variation Δx

$$(\nabla z)_j = \sum_i g'_i(\sum_j f_{j \to i}(x_j)) \sum_j f'_{j \to i}(x_j)$$

- Où chaque $(\nabla z)_j$ est un terme de gradient pour la composante j
- L'idée sera d'exploiter la programmation dynamique pour éviter de recalculer

SIMPLIFIONS LES CHOSES!

- Dans la propagation arrière, nous propageons l'erreur de la fin vers le début grâce à la règle de la chaîne
- Pour notre MLP, nous commençons donc par trouver le vecteur d'erreur suivant:

$$\delta^{(3)} = a^{(3)} - y$$

  - y est le vecteur des étiquettes (ou des vraies classes)
- Ensuite, on calcul l'erreur de la couche cachée:

$$\delta^{(2)} = \left(W^{(2)}\right)^{T} \left(\delta^{(3)} * \frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}} \right)$$

- La dernière partie est la dérivée de la fonction d'activation (sigmoid):

$$\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}} = \left(a^{(2)} * \left(1 - a^{(2)}\right)\right)$$

BACKPROPAGATION

$$\delta^{(2)} = \left(W^{(2)}\right)^T \left(\delta^{(3)} * \frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}\right)$$

- Regardons plus concrètement le calcul
- $W^{(2)}$ est une matrice $t \times h$ (nb classes, nb unités cachées)
- $\delta^{(3)}$ est le vecteur d'erreurs $t \times 1$
- Celui-ci est multipliée par la dérivée $\left(a^{(2)} * \left(1 - a^{(2)}\right)\right)$ qui est un vecteur $t \times 1$ (multiplication par pair d'éléments)

- Donc la matrice transposée résultante $h \times t$ devient un vecteur $h \times 1$ après la multiplication
  - le produit croisé d'une matrice m,n donne le vecteur de taille m
- Le nouveau vecteur d'erreurs $\delta^{(2)}$ est donc de taille $h \times 1$

BACKPROPAGATION

- Lorsque les δ ont été trouvé, nous pouvons retravailler la dérivée de la fonction de coût:

$$\frac{\partial}{\partial w_{i,j}^{l}} J(\mathbf{W}) = a_j^l \delta_i^{(l+1)}$$

- Il faut accumuler la dérivée partielle pour chaque nœud j de la couche l et la $i$ème erreur du nœud au niveau $l + 1$

$$\Delta w_{i,j}^{(l)} := \Delta w_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

- Pour chaque échantillon de l'ensemble d'entraînement!

BACKPROPAGATION

- On peut écrire l'équation précédente sous forme vectorielle (afin d'inclure le calcul pour les échantillons)

$$\Delta W^{(l)} := \Delta^{(l+1)}\left(A^{(l)}\right)^T$$

  - Où $\Delta^{(l+1)}$ est la matrice d'erreurs

- Enfin, nous pouvons régulariser:

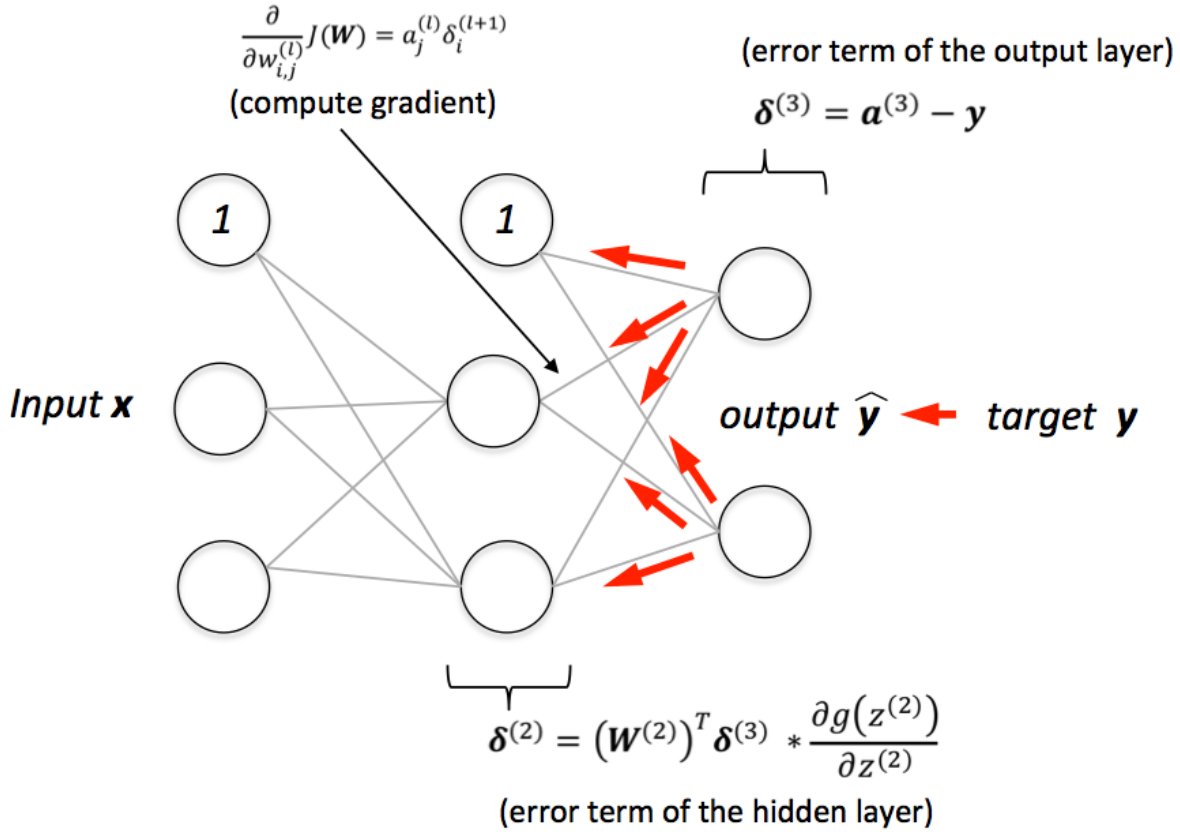
$$\Delta W^{(l)} := \Delta W^{(l)} + \lambda^{(l)}$$

- Finalement, maintenant que nous avons les gradients, il suffit de faire un pas opposé pour mettre à jour les poids:

$$W^{(l)} := W^{(l)} - \eta \Delta W^{(l)}$$

BACKPROPAGATION

- Retour sur le code!



NON-LINÉARITÉ

- Bien que chaque unité du MLP soit indépendamment linéaire, si les fonctions d'activation sont non-linéaires, leur combinaison engendre un modèle non-linéaire
- Si nous utilisons uniquement une activation linéaire, alors peu importe le nombre de couches, le réseau reste toujours un modèle linéaire
  - Et ne gagne pas en expressivité
- Supposons 2 couches linéaires $z^{(2)} = W^{(1)}a^{(1)} + b^{(1)}$ et $z^{(3)} = W^{(2)}z^{(2)} + b^{(2)}$et substituons $z^{(2)}$ dans $z^{(3)}$:

$$z^{(3)} = W^{(2)}(W^{(1)}a^{(1)} + b^{(1)}) + b^{(2)}$$
$$z^{(3)} = W^{(2)}W^{(1)}a^{(1)} + W^{(2)}b^{(1)} + b^{(2)}$$

- On peut écrire cette équation avec un modèle linéaire simple couche:

$$z^{(3)} = W^{(\prime)}a^{(1)} + b^{(\prime)}$$

  - Où $W^{(\prime)} = W^{(2)}W^{(1)}$ et $b^{(\prime)} = W^{(2)}b^{(1)} + b^{(2)}$

APPROXIMATION UNIVERSELLE

- Nous ne pouvons pas faire ça si $a^{(2)} = \frac{1}{1+e^{-z}}$
- De plus, Cybenko G. a montré en 1989 qu'un réseau sigmoïd à propagation avant est un approximateur universel
  - Si le nombre de neurones dans la couche cachée est suffisamment grand
  - Le MLP peut approximer n'importe quelle fonction continue
  - La précision arbitraire dépendrait uniquement de la densité!
- Depuis, la preuve a été généralisée à d'autres fonctions d'activation et à des réseaux de profondeur arbitraire (e.g. ReLU)
- Bon, c'est de la théorie, mais en connaissant cette propriété ça nous donne une idée de la puissance du deep learning!

CONCLUSION

- Pour faire du deep learning, il faut:
  - Des fonctions **différentiables**
  - Un algorithme de calcul de gradient efficace: **backpropagation**
  - Des fonctions d'activation **non-linéaires**
  - Un **optimiseur** itératif de paramètres

… et surtout une **quantité importante** de données!!!

RÉFÉRENCES ORIGINALES

MLP	
MNIST	
Backpropagation	
Approximation sigmoïd	Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. Mathematics of control, signals and systems, 2(4), 303-314.
Approximation universelle	Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. Neural networks, 4(2), 251-257.