IT496: Introduction to Data Mining

Lecture 27-28

Training DNN: Vanishing Gradient Problem

(Slides are created from the lecture notes of Dr. Derek Bridge, UCC, Ireland)

Arpit Rana

20th October 2023

Problems in Training Deep Neural Network

- ***Vanishing Gradient Problem***
  The gradients grow smaller and smaller when flowing backward through the DNN while training.

- ***Need enough (labelled) training data***
  Large network demands huge amount of data or it might be too costly to label.

- ***Training may be extremely slow***
  Training very large DNN using (batch) gradient descent can be painfully slow.

- ***Overfitting***
  A model with millions of parameters would severely risk overfitting the training data.

Problems and Solutions in Training Deep Neural Network

- ***Vanishing Gradient Problem***
  - Better Initialization
  - Non-saturating Activation Functions
  - Batch Normalization
- ***Need enough (labelled) training data***
  - Reusing pretrained layers (transfer learning)
  - Unsupervised pre-training
  - Pre-training on an auxiliary task
- ***Training may be extremely slow***
  - Using faster optimizers
  - Learning rate scheduling
- ***Overfitting***
  - Reducing the network size
  - Weight regularization
  - Dropout
  - Early stopping

Problems and Solutions in Training Deep Neural Network

- ***Vanishing Gradient Problem***
    - Better Initialization
    - Non-saturating Activation Functions
    - Batch Normalization
- ***Need enough (labelled) training data***
    - Reusing pretrained layers (transfer learning)
    - Unsupervised pre-training
    - Pre-training on an auxiliary task
- ***Training may be extremely slow***
    - Using faster optimizers
    - Learning rate scheduling
- ***Overfitting***
    - Reducing the network size
    - Weight regularization
    - Dropout
    - Early stopping

These will not be covered in this course.

This will be covered in the next lecture.

Vanishing Gradient Problem

Activation Function: sigmoid (logistic function)

Weight Initialization Technique: random initialization (using a normal distribution with a mean 0 and a standard deviation of 1).

X. Glorot and Y. Bengio found that this combination leads to the following.

- The variance of the outputs of each layer is much greater than the variance of its inputs.
- The variance keeps increasing after each layer until the activation function saturates at the top layers.

The logistic function has a mean of 0.5, not 0.

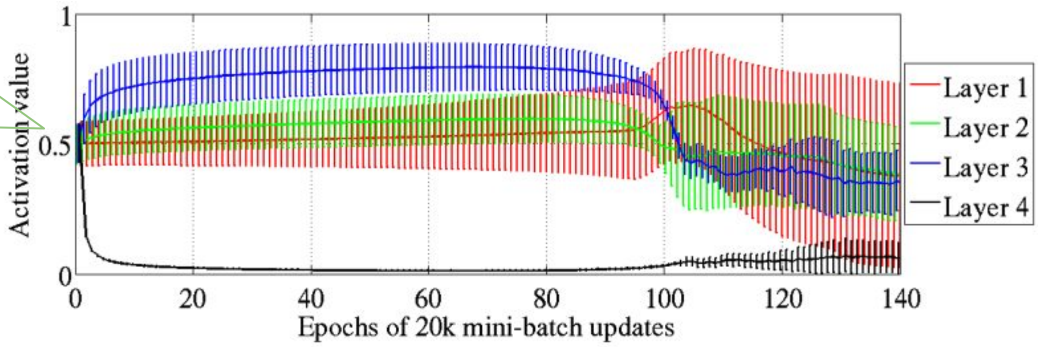
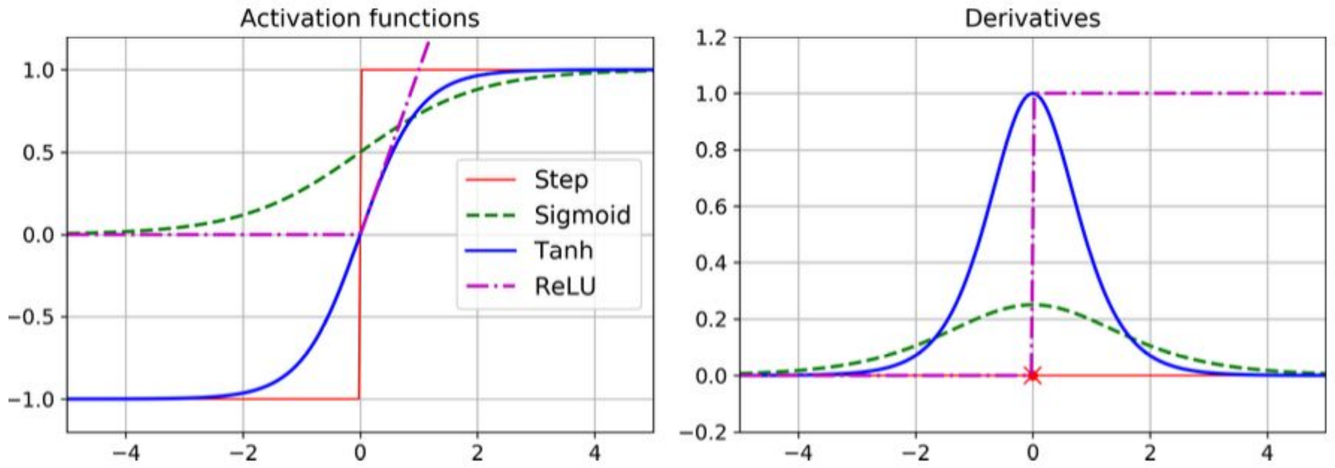


Image Source: "Understanding the Difficulty of Training Deep Feedforward Neural Networks," X. Glorot, Y Bengio (2010).

Vanishing Gradient Problem

Looking at the logistic function and its derivative plots below.

- When inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0.



- Thus, during backpropagation, there's not much gradient to propagate back to earlier layers (and what little gradient there is, gets diluted as it propagates back).

Vanishing Gradient Problem

- Each weight is updated by an amount proportional to the gradient of the loss function with respect to that weight.
  - But if the gradient is very small, the weight doesn't change much.
    - This may prevent the network from converging to a good approximation of the target function.
  - We can now see that this is worse for deeper networks.
    - The error signal becomes ever smaller as it is back propagated.
- We look at the following solutions:
  - Better weight initialization;
  - Non-saturating activation functions;
  - Batch normalization.

Weight Initialization

Glorot and Bengio (2010) argued that -

- the variance of the outputs of each layer should be equal to the variance of its inputs (going forward), and

- the gradients should have equal variance before and after flowing through a layer (going backward).

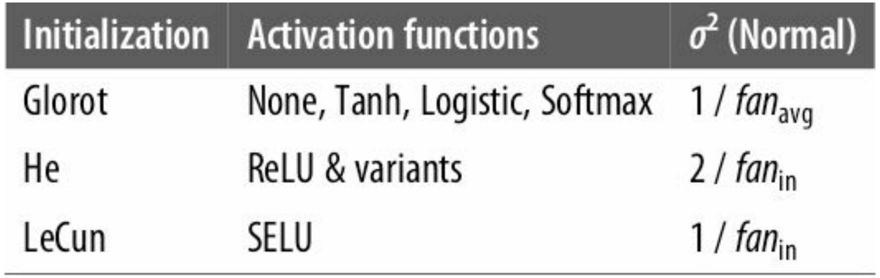
Both the conditions are not guaranteed unless the number of input units to a layer (fan_{in}) equals the number of neurons, i.e., number of output units (fan_{out})

Weight Initialization

- Glorot (Xavier) Initialization: connection weights must be initialized randomly while using the logistic activation function -
    - Normal distribution with mean 0 and variance $\sigma^2 = 1 / fan_{avg}$
    - Or a uniform distribution between −r and + r, with r = sqrt (3 * fan_{avg})

    where, $fan_{avg} = (fan_{in} + fan_{out}) / 2$

- Other similar strategies for different activation functions were proposed -

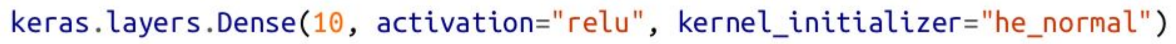


Initialization	Activation functions	σ^2 (Normal)
Glorot	None, Tanh, Logistic, Softmax	$1 / fan_{avg}$
He	ReLU & variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Weight Initialization in Keras

- By default, Keras uses *Glorot initialization* with a *uniform* distribution.

- You can change this to *He* initialization by setting `kernel_initializer="he_uniform"` or `kernel_initializer="he_normal"` when creating a layer, like this:



```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Non-Saturating Activation Functions

Certain activation functions, including the sigmoid function, are one cause of the vanishing gradient problem.

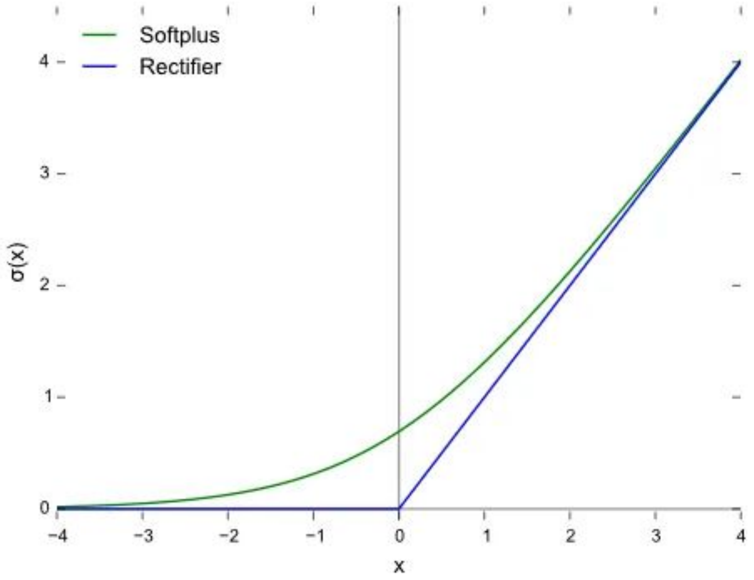
- When the input to this function becomes large (positive or negative), the function saturates (i.e. becomes very flat).
    - Even when the gradient is at its greatest (when input z is 0 and $\boldsymbol{\sigma}(z) = 0.5$), it is only 0.25 (gradient = $\boldsymbol{\sigma}(z)(1 - \boldsymbol{\sigma}(z))$).
        - So in the back propagation, gradients always diminish by a quarter or more.
- This is why we rarely use the sigmoid function as the activation function in the hidden layers of deep networks.

Non-Saturating Activation Functions

- Lots of alternatives have been proposed, including the ReLU (***rectified linear unit***) activation function,
- ReLU does not saturate for positive values and it is quite fast to compute.

$$ReLU(z) = \max(0, z)$$

$$Softplus(z) = log(1 + e^z)$$



ReLU and its smooth variant, ***softplus***.

Non-Saturating Activation Functions

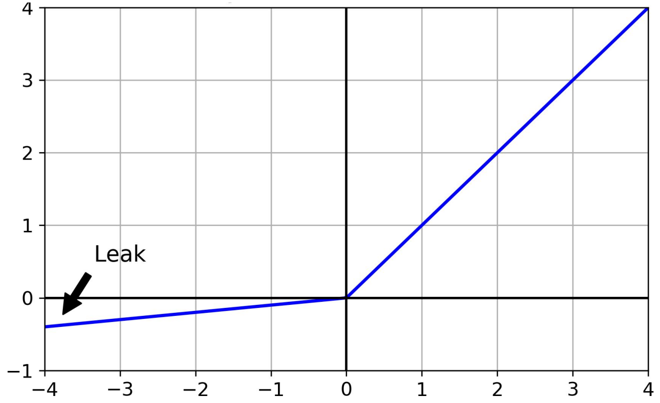
- Neurons that use the ReLU activation function have obvious problems too:
    - If their input (weighted sum) is negative, the output is zero; and the gradient is zero; and
    - if this is true for all examples in the training set then, in effect, the neuron dies. This is known as “***dying ReLU***” problem.
    - The gradient changes abruptly at z = 0, which can make Gradient Descent bounce around.
- Despite its problems, it remains a popular choice.

Non-Saturating Activation Functions

Leaky ReLU have been proposed having at least some non-zero gradient for negative inputs (alternatives to ReLU are slower to compute and they introduce further hyperparameters.)

The hyperparameter α defines how much the function "leaks": it is the slope of the function for z < 0, and is typically set to 0.01.

$$LeakyReLU_{\alpha}(z) = \max(\alpha z, z)$$



```
layers.Dense(7 * 7 * 128),
layers.LeakyReLU(alpha=0.2),
```

Instead of being a hyperparameter, α becomes a parameter that can be modified by backpropagation like any other parameter, this version of ReLU is known as *Parametric Leaky ReLU (PReLU)*.

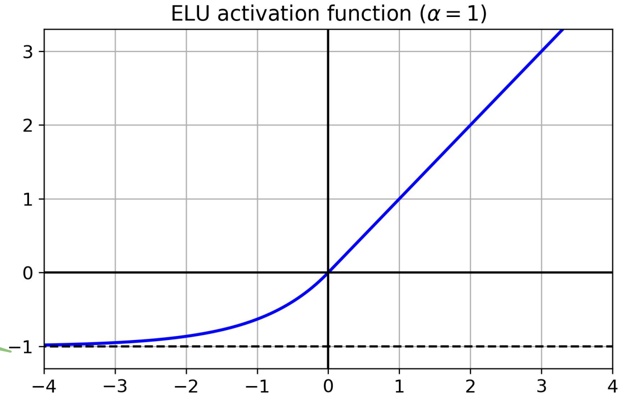
Non-Saturating Activation Functions

ELU (Exponential Linear Unit)

- It helps alleviate the vanishing gradient problem.
- It has a non-zero gradient for z < 0, which avoids the dead neurons problem.
- If α = 1, then the function is smooth everywhere, which helps speed up gradient descent (does not bounce much left and right of z = 0.)

$$ELU_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & if\, z < 0 \\ z & if\, z \geq 0 \end{cases}$$

The hyperparameter α defines the value that the ELU function approaches when z is a large negative number.



Non-Saturating Activation Functions

SELU (Scaled Exponential Linear Unit)

- If we build a neural network with a stack of dense layers, and all the hidden layers use SELU activation, then the network will "*self-normalize*" (the output of each layer will tend to preserve mean 0 and standard deviation 1 during training).
- The input features must be standardized (mean 0 and standard deviation 1).
- Every hidden layer weights must also be initialized using LeCun normal initialization. In Keras, this means setting `kernel_initializer="lecun_normal"`.

$$SELU(z) = \begin{cases} \lambda\alpha(\exp(z) - 1) & if\, z < 0 \\ \lambda z & if\, z \geq 0 \end{cases}$$

[α **(>=1)** and λ **(>1)** are predefined constants.]

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

Non-Saturating Activation Functions

Which activation function should we use for the hidden layers of our deep neural network?

- In general, ***SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic***
- If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$).
- If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may just use the default α values used by Keras (e.g., 0.3 for the leaky ReLU).
- If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular PReLU if you have a huge training set.

Batch Normalization

We previously studied the usefulness of feature scaling when doing Gradient Descent for, e.g., linear regression.

… and we've been doing this to the features in our neural networks too.

- But, if this is a good idea for the inputs to the first hidden layer, why not use the same idea for the inputs to subsequent layers?
- In other words, we normalize the activations (outputs) of layer l prior to them being used as inputs to layer $l + 1$.
- This will control the distribution of the values throughout the training process.
- This, in essence, is the idea of ***batch normalization.***

Batch Normalization

In summary, for a given layer, it standardizes the outputs of the neurons (subtract the mean, divide by the standard deviation), ***then it scales the result and adds an offset***.

Batch Normalization Algorithm:

1. $$\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

2. $$\boldsymbol{\sigma}_B{}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \boldsymbol{\mu}_B\right)^2$$

3. $$\widehat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B{}^2 + \epsilon}}$$

4. $$\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \widehat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$$

During training, four parameter vectors are learned in each batch-normalized layer:

- γ (the output scale vector) and β (the output offset vector) are learned through regular backpropagation, and
- μ (the final input mean vector), and σ (the final input standard deviation vector) are estimated using an exponential moving average.

What about the test time?

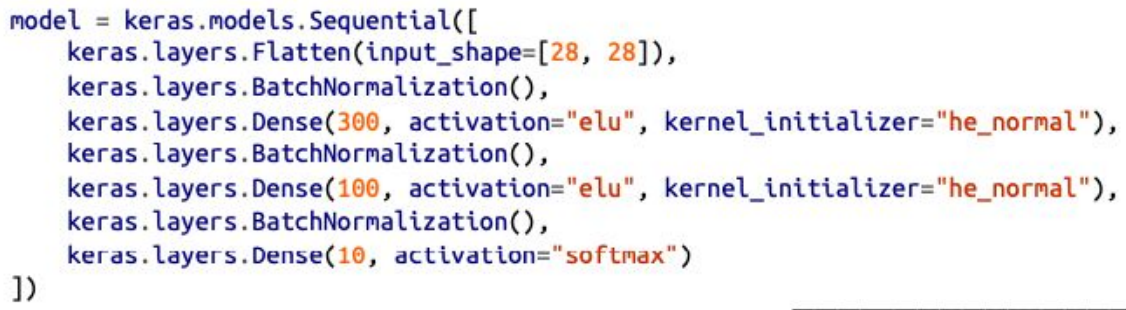
Batch Normalization

- Batch Normalization reduces the vanishing gradients problem so much, we can even use saturating activation functions.
- Training becomes less sensitive to the method used for randomly initializing weights.
- Much larger learning rates work (faster convergence) with less risk of divergence.
- It acts like a regularizer.

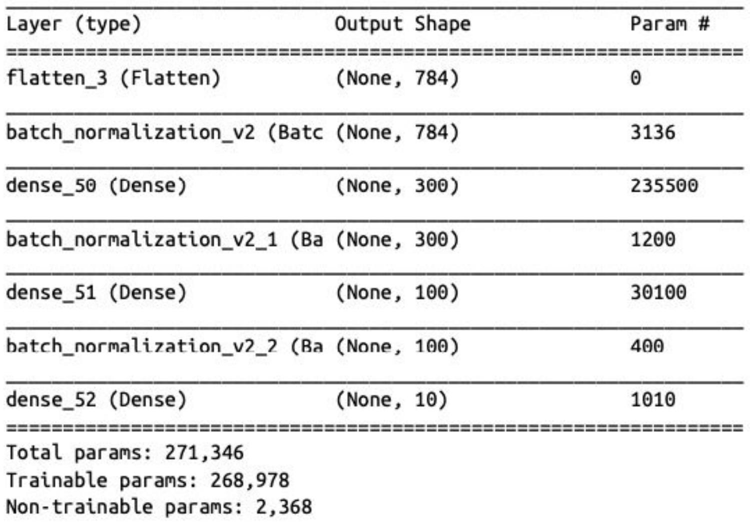
Batch Normalization in Keras

- Just add another layer!
- For example,
  - `x = Dense(512, activation="relu")(x)`
  - `x = BatchNormalization()(x)`
- This is how we will do batch normalization in this course.
- Ignore: in fact, there is some debate about whether we should batch normalize the activations of the layer (as above) or the weighted sum, before applying the activation function (as below):
  - `x = Dense(512, activation="linear", use_bias=False)(x)`
  - `x = BatchNormalization()(x)`
  - `x = Activation("relu")(x)`
- We will stick with the former, which is more concise.

Batch Normalization in Keras



```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
batch_normalization_v2 (Batc (None, 784)               3136
_________________________________________________________________
dense_50 (Dense)             (None, 300)               235500
_________________________________________________________________
batch_normalization_v2_1 (Ba (None, 300)               1200
_________________________________________________________________
dense_51 (Dense)             (None, 100)               30100
_________________________________________________________________
batch_normalization_v2_2 (Ba (None, 100)               400
_________________________________________________________________
dense_52 (Dense)             (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
```

Exploring Vanishing Gradient with MNIST

```
def build_mnist_network(activation, initializer, use_batch_norm):
    inputs = Input(shape=(28 * 28,))
    x = Rescaling(scale=1./255)(inputs)
    x = Dense(512, activation=activation, kernel_initializer=initializer)(x)
    if use_batch_norm:
        x = BatchNormalization()(x)
    outputs = Dense(10, activation="softmax", kernel_initializer=initializer)(x)
    model = Model(inputs, outputs)
    model.compile(optimizer=RMSprop(learning_rate=0.0001), loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return model
```

```
networks = [
    build_mnist_network("sigmoid", "random_normal", False),
    build_mnist_network("sigmoid", "random_normal", True),
    build_mnist_network("sigmoid", "glorot_uniform", False),
    build_mnist_network("sigmoid", "glorot_uniform", True),
    build_mnist_network("relu", "random_normal", False),
    build_mnist_network("relu", "random_normal", True),
    build_mnist_network("relu", "glorot_uniform", False),
    build_mnist_network("relu", "glorot_uniform", True)
]

for network in networks:
    network.fit(mnist_x_train, mnist_y_train, epochs=10, batch_size=32, verbose=0)
    test_loss, test_acc = network.evaluate(mnist_x_test, mnist_y_test, verbose=0)
    print(test_acc)
```

Output:

```
0.9451000094413757
0.9549999833106995
0.9435999989509583
0.9496999979019165
0.9746999740600586
0.9814000129699707
0.9746999740600586
0.9840999841690063
```

Hyperparameter Tuning

- First an observation about validation sets:
  - When using neural networks, we typically have a large dataset.
  - Hence, we use holdout to split the dataset into train, validation and test sets.
  - If you have a smaller dataset, where you can afford to split off a test set, but you cannot afford to split off a validation set, then you might want to use k-fold cross-validation, as we did in our scikit-learn examples.
  - Keras does not have any in-built cross-validation functions. You'd have to write your own.

Hyperparameter Tuning

- Second, an observation about hyperparameter tuning:
    - Hyperparameter tuning involves training lots of different models with different values for the hyperparamaters, and comparing their performance on the validation data.
    - This is often a problem with neural networks: training can be slow (due to large datasets and models that have many parameters) and so training lots of different models for hyperparameter tuning is very time-consuming.
    - People often don't bother! They just pick hyperparameter values out of their heads, or accept the Keras defaults.
    - If you can afford to be more systematic, there is a separate library (which you would need to install) called `KerasTuner`. It automates hyperparameter tuning, similar to what we saw in scikit-learn.

Next lecture

Overfitting in Neural Network

20th October 2023