S03.1: Aprendizaje Profundo II

Redes Neuronales y *Deep Learning*

Dr. Juan Bekios Calfa

Magíster en Inteligencia Artificial

UNIVERSIDAD ADOLFO IBÁÑEZ

Información de Contacto

- Juan Bekios Calfa
  - email: juan.bekios@edu.uai.cl, juan.bekios@ucn.cl
  - Web page: http://jbekios.ucn.cl
  - Teléfono: 235(5162) - 235(5125)

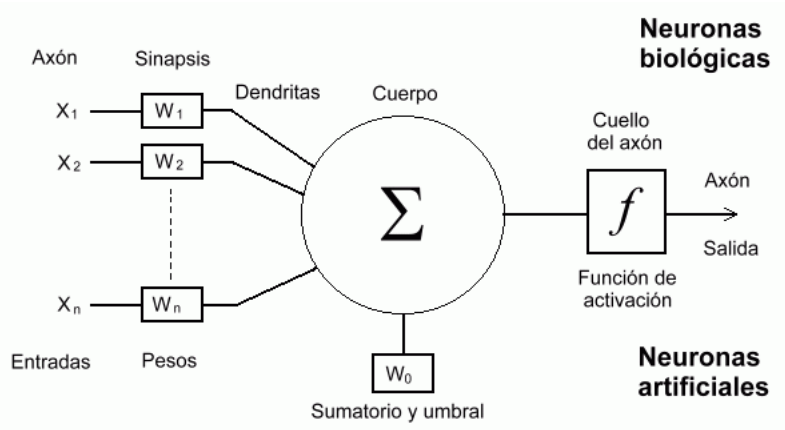
UNIVERSIDAD ADOLFO IBÁÑEZ

Contenidos

UNIVERSIDAD ADOLFO IBÁÑEZ

Contenidos

Modelo de una neurona simple



https://www.um.es/LEQ/Atmosferas/Ch-VI-3/F63s4p3.htm

Redes neuronales profundas

Funciones de Activación

Análisis y tipos de funciones de activación

Funciones de activación

- Las **funciones de activación** son una parte crucial de los modelos de **aprendizaje profundo**, ya que agregan la **no linealidad** a las redes neuronales.
- Existe una gran variedad de **funciones de activación** en la literatura, y algunas son más beneficiosas que otras.
- Debemos entender la importancia de elegir una buena **función de activación**, y qué problemas pueden ocurrir si no lo hacemos.

Escribir una función de activación (Pytorch)

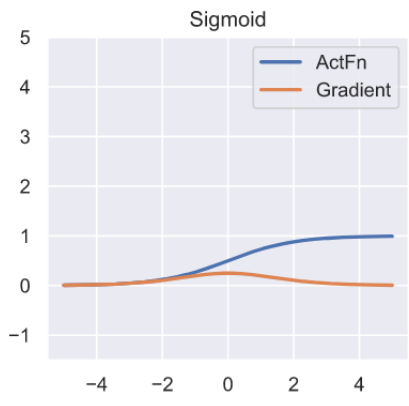
- En Pytorch existen **funciones de activación** definidas en el paquete torch.nn.
- Para entender el funcionamiento de las **funciones de activación** en Pytorch, implementaremos las más comunes.

```
class FuncionDeActivacion(nn.Module):

    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__
        self.config = {"Nombre de la clase": self.name}
```

Funciones de activación I

Sigmoide:

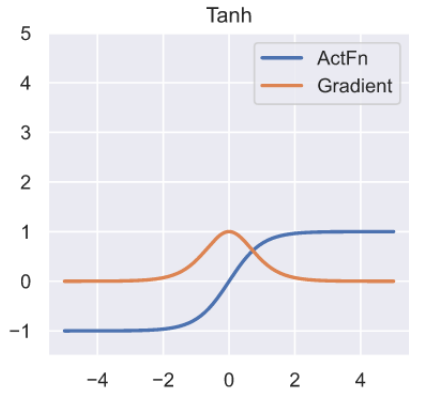
```
1  class Sigmoide(FuncionDeActivacion):
2      def forward(self, x):
3          return 1 / (1 + torch.exp(-x))
```



Funciones de activación II

Tangente Hiperbólica:

```
1  class Tanh(FuncionDeActivacion):
2      def forward(self, x):
3          x_exp, neg_x_exp = torch.exp(x), torch.
               exp(-x)
4          return (x_exp - neg_x_exp) / (x_exp +
               neg_x_exp)
```



Funciones tipo ReLU (I)

- Una de las **funciones de activación** más populares para redes más profundas es la **Unidad Lineal Rectificada** (**ReLU**).
- La función de activación ReLU, tiene como **beneficio** importante que mantiene un gradiente grande para una amplia gama de valores.
- Con base en la función **ReLU** se han propuesto **muchas variaciones**: LeakyReLU, ELU, Swish, etc.

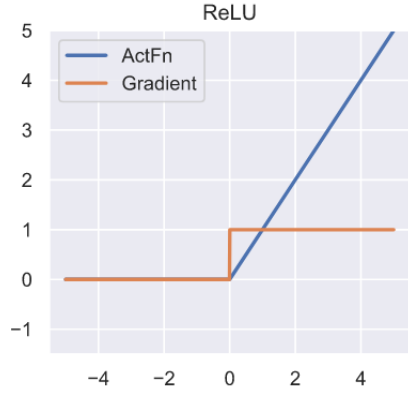
Funciones tipo ReLU (II)

- **Función de activación ELU**: Reemplaza la parte negativa con un decaimiento exponencial.
- **Función de activación Swish**: Es suave y no monótono (es decir, contiene un cambio de signo en el gradiente).
  Se ha demostrado que esto previene las **neuronas muertas** como en la activación ReLU estándar, especialmente para redes profundas.
  Esta función es el resultado de un gran experimento para **encontrar** una **función de activación óptima**.

Funciones de activación III

Unidad Lineal Rectificada (**ReLU**):

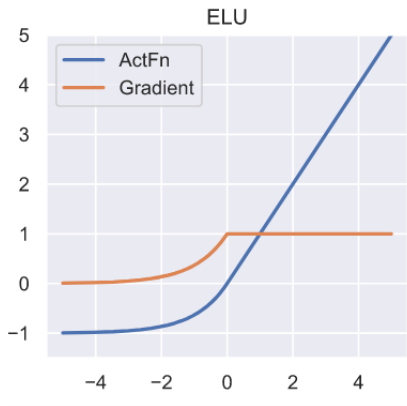
```
1  class ReLU(FuncionDeActivacion):
2      def forward(self, x):
3          return x * (x > 0).float()
```



Funciones de activación IV

ELU:

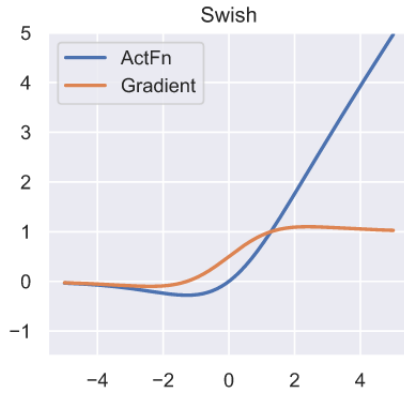
```
1  class ELU(FuncionDeActivacion):
2      def forward(self, x):
3          return torch.where(x > 0, x, torch.exp(x)
               -1)
```



Funciones de activación V

Swish:

```
class Swish(ActivationFunction):
    def forward(self, x):
        return x * torch.sigmoid(x)
```



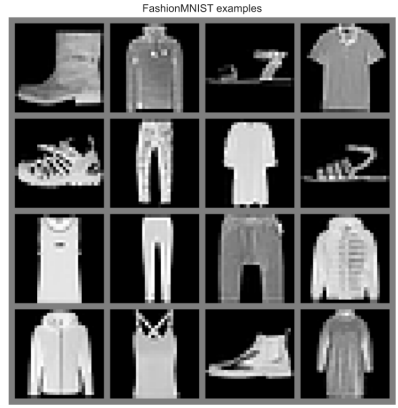
UNIVERSIDAD ADOLFO IBÁÑEZ

Cálculo de gradientes

```python
def get_grads(act_fn, x):
    """
    Calcula los gradientes de una función de activación.
    Inputs:
        act_fn - Un objeto FuncionDeActivacion con el método forward implementado.
        x - 1D tensor de entrada.
    Output:
        Un tensor de la misma dimension de x con los gradientes de la función
            act_fn.
    """
    x = x.clone().requires_grad_() # Clonamos y declaramos cálculo de gradientes
    out = act_fn(x)
    out.sum().backward() # Sumar los resultados para calcular los gradientes
    return x.grad # Obtener los gradientes por "x.grad"
```

Análisis de funciones de activación

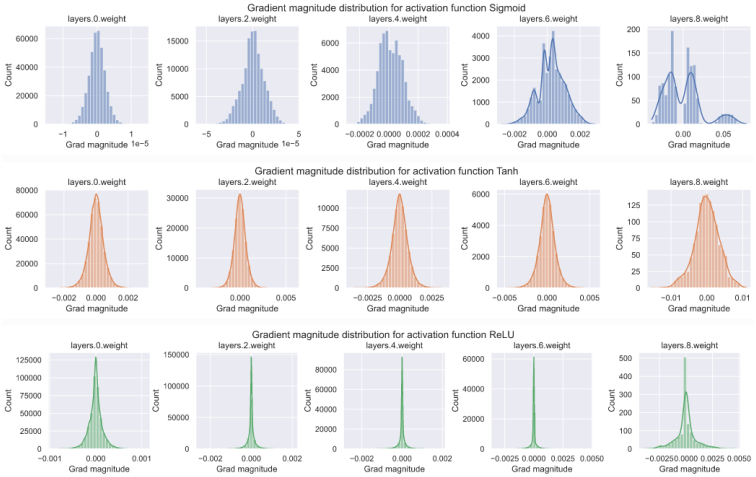
- Realizaremos un análisis del comportamiento de una red utilizando datos de la base de datos MNIST.
- Se configurará un red base de cuatro capas ocultas para analizar el comportamiento de las funciones de activación en el desempeño de la red.



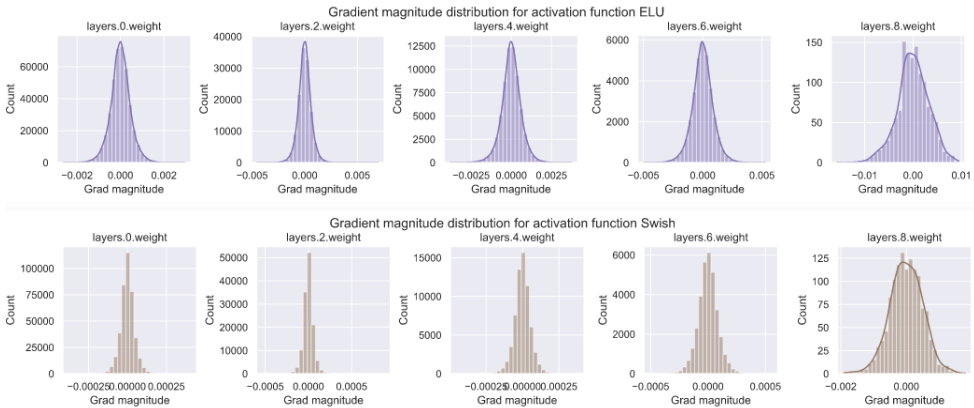
Modelo de red neuronal

```python
class BaseNetwork(nn.Module):
    def __init__(self, act_fn, input_size=784, num_classes=10, hidden_sizes=[512, 256, 256, 128]):
        super().__init__()
        layers = []
        layer_sizes = [input_size] + hidden_sizes
        for layer_index in range(1, len(layer_sizes)):
            layers += [nn.Linear(layer_sizes[layer_index-1], layer_sizes[layer_index]), act_fn]
        layers += [nn.Linear(layer_sizes[-1], num_classes)]
        self.layers = nn.Sequential(*layers)
        self.config = {"act_fn": act_fn.config, "input_size": input_size, "num_classes": num_classes, "hidden_sizes": hidden_sizes}
    def forward(self, x):
        x = x.view(x.size(0), -1) # Aplanar vector de entrada
        out = self.layers(x)
        return out
```

Flujo de gradientes en inicialización (I)



Flujo de gradientes en inicialización (II)

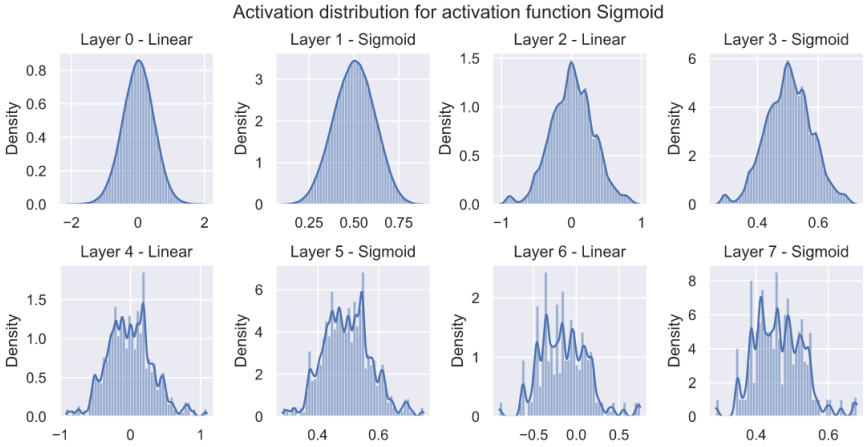


Gradientes

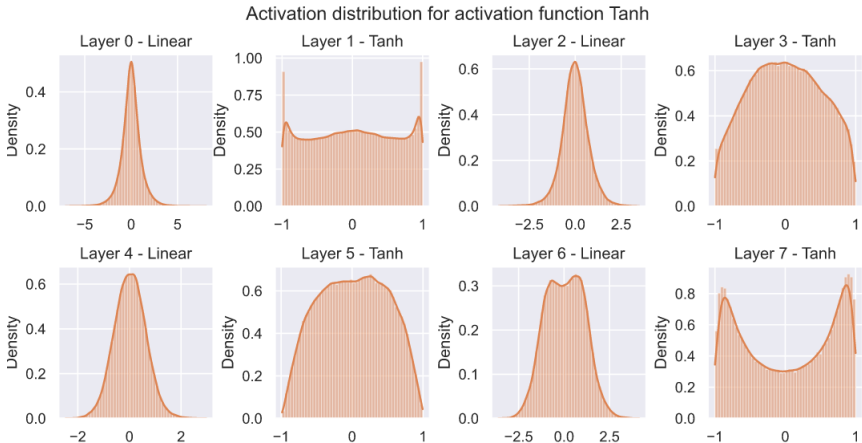
- Si el gradiente a través de la función de activación es considerablemente más pequeño que 1, nuestros gradientes se desvanecerán hasta que lleguen a la capa de entrada.
- Si el gradiente a través de la función de activación es mayor que 1, los gradientes aumentan exponencialmente y pueden explotar.

UNIVERSIDAD ADOLFO IBÁÑEZ

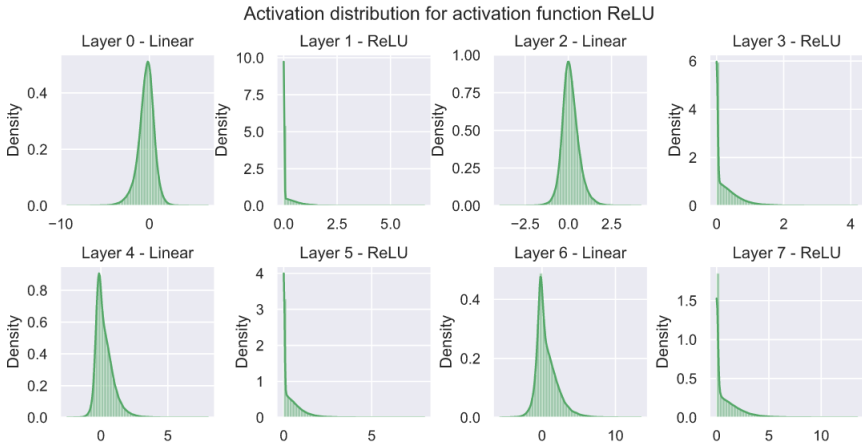
Función de activación después del entrenamiento (I)



Función de activación después del entrenamiento (II)

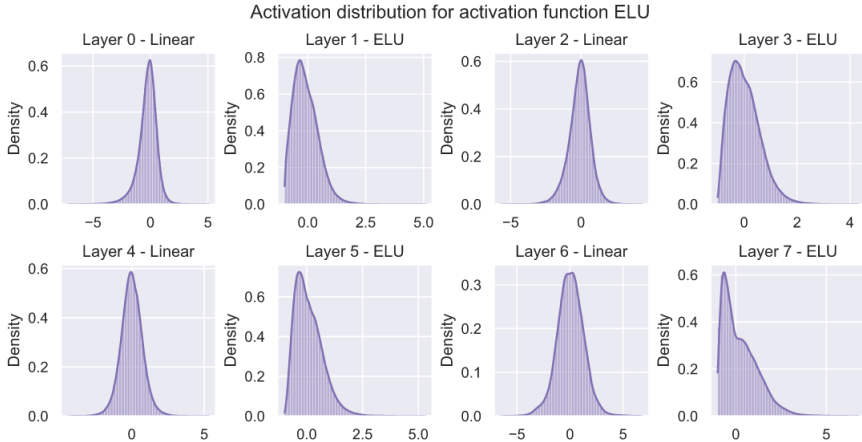


Función de activación después del entrenamiento (III)

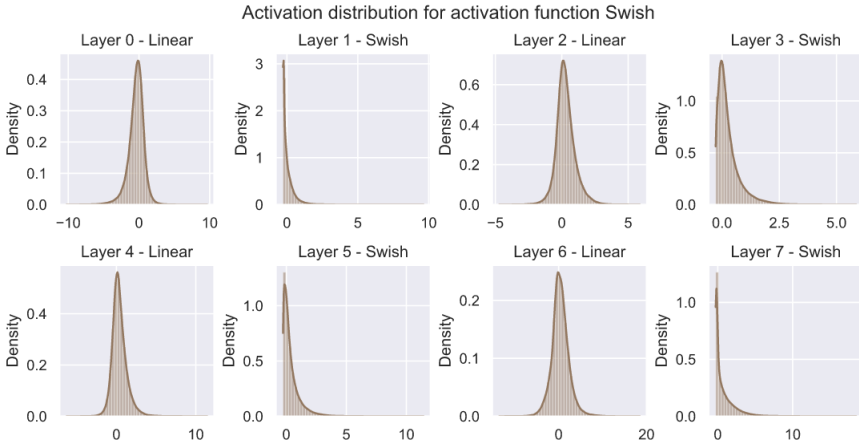


UNIVERSIDAD ADOLFO IBÁÑEZ

Función de activación después del entrenamiento (IV)



Función de activación después del entrenamiento (V)



Inicialización del modelo

Algoritmos de inicialización

Inicialización

- Al aumentar la **profundidad de las redes neuronales**, existen varios desafíos a los que nos enfrentamos para lograr un buen entrenamiento y su convergencia.
- Debemos mantener un **flujo de gradiente** estable a través de la red, ya que, de lo contrario, podríamos encontrar gradientes que se desvanecen o explotan.

Propiedades deseables

- La **varianza de la entrada** debe propagarse a través del modelo hasta la **última capa**, de modo que tengamos una desviación estándar similar para las neuronas de salida.
- Si la **varianza desapareciera**, cuanto más profundizamos en nuestro modelo, se vuelve mucho más **difícil optimizar el modelo**, ya que la entrada a la siguiente capa es básicamente un único valor constante.
- Si la **varianza aumenta**, es probable que explote (es decir, se vaya al infinito) cuanto más profundo diseñemos nuestro modelo.
- Si la **distribución de gradiente** con la misma variabilidad entre las capas. Si la primera capa recibe gradientes mucho más pequeños que la última capa, tendremos dificultades para elegir una **tasa de aprendizaje** adecuada.

Análisis de la inicialización

- Para el análisis utilizaremos la función de activación identidad o lineal:

```
    class Identity(nn.Module):
        def forward(self, x):
            return x
```

- Construiremos un modelo de red neuronal artificial utilizando la clase definida anteriormente:

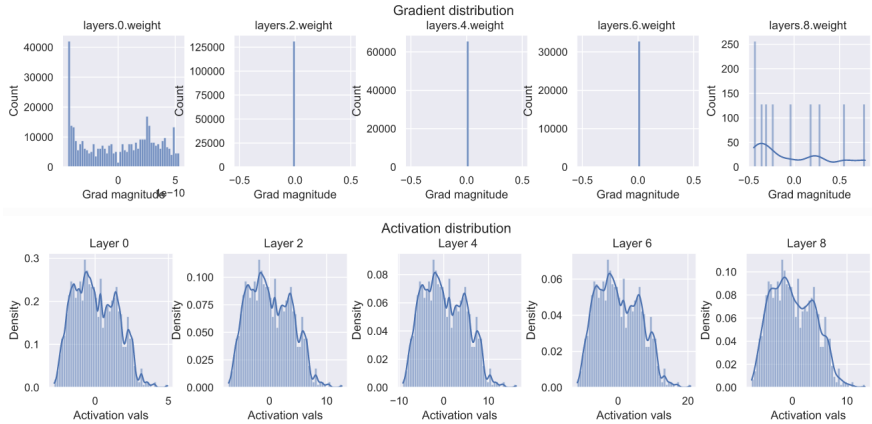
```
    model = BaseNetwork(act_fn=Identity()).to(device)
```

Inicialización: Constante (I)

- La primera inicialización que podemos considerar es inicializar todos los pesos con el mismo valor constante.
- Intuitivamente, establecer todos los pesos en cero no es una buena idea, ya que el gradiente propagado será cero.
- Para realizar una prueba inicializaremos todos los pesos sinápticos en 0.005.

```
1       def const_init(model, c=0.0):
2           for name, param in model.named_parameters():
3               param.data.fill_(c)
4
5       const_init(model, c=0.005)
```

Inicialización: Constante (II)



Inicialización: Constante (III)

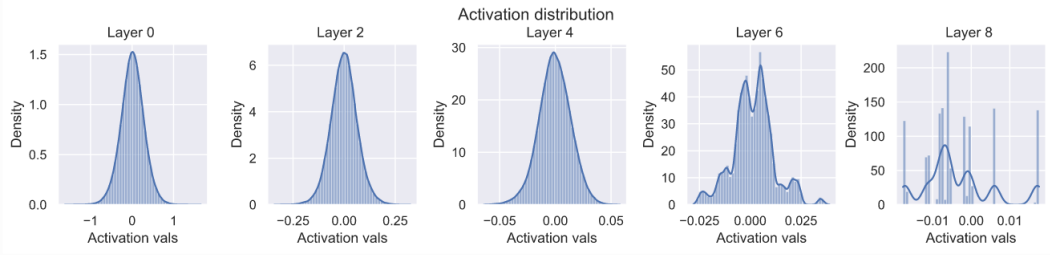
- Solo la primera y la última capa tienen distribuciones de gradientes diversas.
- Las otras tres capas tienen el mismo gradiente para todos los pesos (valores muy cercanos a cero).
- Gradiente en cero, o cercanos, para los parámetros que se han inicializado con los mismos valores, significa que siempre tendremos el mismo valor para esos parámetros.
- No se debe entrenar una red con inicialización constante.

Inicialización: Varianza constante (I)

- Inicializamos los parámetros mediante un muestreo aleatorio de una distribución como una gaussiana.

```
    def var_init(model, std=0.01):
        for name, param in model.named_parameters():
            param.data.normal_(std=std)

    var_init(model, std=0.01)
```

Inicialización: Varianza constante (II)



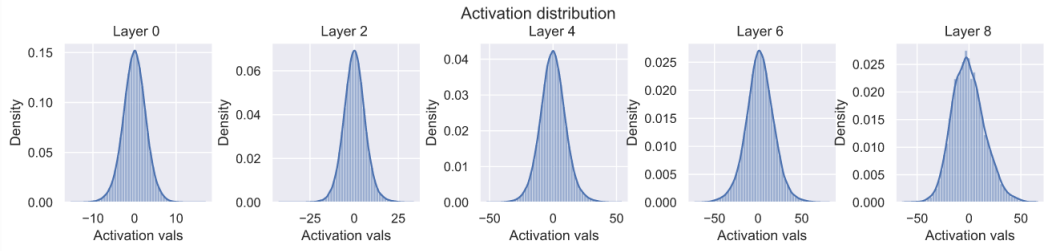

```
Layer 0 - Variance: 0.077930
Layer 2 - Variance: 0.004179
Layer 4 - Variance: 0.000207
Layer 6 - Variance: 0.000102
Layer 8 - Variance: 0.000083
```

Inicialización: Varianza constante (III)

- Cambiamos el valor de la varianza a una más grande.

```
var_init(model, std=0.1)
```

Inicialización: Varianza constante (IV)




```
Layer 0 - Variance: 7.901733
Layer 2 - Variance: 38.941471
Layer 4 - Variance: 103.246284
Layer 6 - Variance: 262.478546
Layer 8 - Variance: 248.354202
```

Inicialización: Varianza constante (V)

- Con una desviación estándar más alta, es probable que las activaciones exploten.
- Se podría jugar con los valores de desviación estándar específicos, pero será difícil encontrar uno que nos brinde una buena distribución de activación entre capas y que sea muy específico para nuestro modelo.
- También se podría probar con el número de capas ocultas, o por capa, pero no sería ni recomendable y/o eficiente.

Inicialización: Valores apropiados (I)

Hemos visto que necesitamos **muestrear** los pesos de una **distribución**, pero no estamos seguros de cuál exactamente.
Para encontrar una optimización óptima, establecemos dos requisitos:

- La media de las activaciones debe ser cero.
- La variación de las activaciones debe permanecer igual en todas las capas.

Inicialización: Valores apropiados (II)

$$\mathsf{Var}(X \cdot Y) = \mathbb{E}(Y)^2\mathsf{Var}(X) + \mathbb{E}(X)^2\mathsf{Var}(Y) + \mathsf{Var}(X)\mathsf{Var}(Y) = \mathbb{E}(Y^2)\mathbb{E}(X^2) - \mathbb{E}(Y)^2\mathbb{E}(X)^2$$

$$y_i = \sum_j w_{ij} x_j \qquad \text{Cálculo neurona sin bias}$$

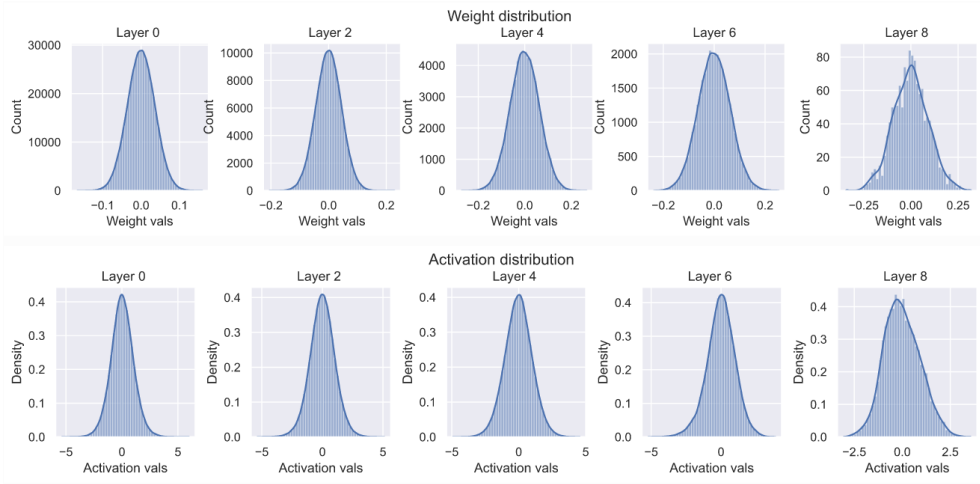
$$\begin{aligned}
\mathsf{Var}(y_i) = \sigma_x^2 &= \mathsf{Var}\left(\sum_j w_{ij} x_j\right) \\
&= \sum_j \mathsf{Var}(w_{ij} x_j) && \text{La entrada de datos } (x) \text{ es independiente de los pesos } (w) \\
&= \sum_j \mathsf{Var}(w_{ij}) \cdot \mathsf{Var}(x_j) && \text{Regla de la varianza con esperanzas en 0} \\
&= d_x \cdot \mathsf{Var}(w_{ij}) \cdot \mathsf{Var}(x_j) && \text{Varianza igual en todas las capas } d_x \text{ elements} \\
&= \sigma_x^2 \cdot d_x \cdot \mathsf{Var}(w_{ij}) \\
\Rightarrow \mathsf{Var}(w_{ij}) = \sigma_W^2 &= \frac{1}{d_x}
\end{aligned}$$

Inicialización: Valores apropiados (I)

- Por lo tanto, deberíamos inicializar la distribución de pesos con una varianza de la inversa de la dimensión de entrada d_x. Implementémoslo a continuación y verifiquemos si esto se cumple:

```
def equal_var_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            param.data.normal_(std=1.0/math.sqrt(param.shape[1]))

equal_var_init(model)
```

Inicialización: Valores apropiados (IV)



Inicialización: Valores apropiados (V)

- La varianza se mantiene constante en todas las capas.
- La inicialización no nos restringe a una distribución normal, pero permite cualquier otra distribución con una media de 0 y una varianza de $1/d_x$.

Inicialización: Xavier (I)

Considera la media armónica entre la entrada y la salida.

$$W \sim \mathcal{N}\left(0, \frac{2}{d_x + d_y}\right)$$

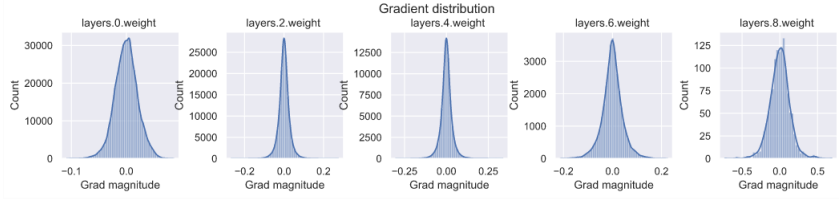
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{d_x + d_y}}, \frac{\sqrt{6}}{\sqrt{d_x + d_y}}\right]$$

Inicialización: Xavier (II)

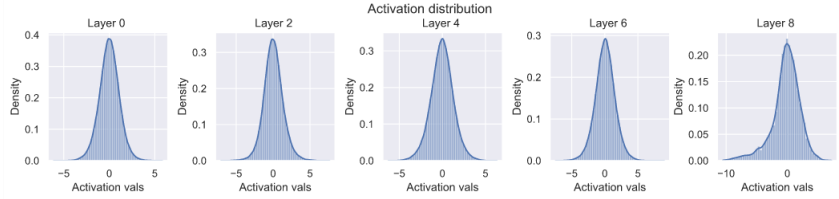
- Implementación de la inicialización Xavier.

```
def xavier_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        else:
            bound = math.sqrt(6)/math.sqrt(param.shape[0]+param.shape[1])
            param.data.uniform_(-bound, bound)

xavier_init(model)
```

Inicialización: Xavier (III)




```
capas.0.peso - Variación: 0.000436
capas.2.peso - Variación: 0.000747
capas.4.peso - Variación: 0.001149
capas.6.peso - Variación: 0.001744
capas.8.peso - Variación: 0.017655
```




```
Capa 0 - Varianza: 1.216592
Capa 2 - Variación: 1.719161
```

Inicialización: Xavier (IV)

- La **inicialización de Xavier** equilibra la variación de gradientes y activaciones.
- La inicialización de Xavier funciona con funciones de **activación lineales** (como el ejemplo) y con **funciones de activación no lineales**.

Inicialización: Kaiming (I)

En Xavier se asume que las no linealidades en las últimas capas se suponen continuas. ¿Qué sucede en el caso de la función de activación ReLU?

$$\mathsf{Var}(w_{ij}x_j) = \underbrace{\mathbb{E}[w_{ij}^2]}_{=\mathsf{Var}(w_{ij})}\ \mathbb{E}[x_j^2] - \underbrace{\mathbb{E}[w_{ij}]^2}_{=0}\mathbb{E}[x_j]^2 = \mathsf{Var}(w_{ij})\mathbb{E}[x_j^2]$$

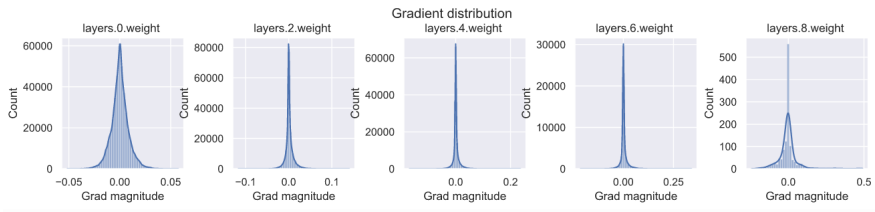
$$\begin{aligned}
\mathbb{E}[x^2] &= \mathbb{E}[\max(0,\tilde{y})^2] \\
&= \frac{1}{2}\mathbb{E}[\tilde{y}^2] && \tilde{y} \text{ centrada en cero y simétrica} \\
&= \frac{1}{2}\mathsf{Var}(\tilde{y})
\end{aligned}$$

Inicialización: Kaiming (II)

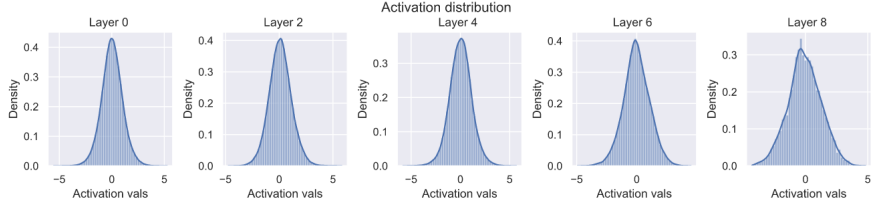
- Implementación de la inicialización Kaiming.

```
def kaiming_init(model):
    for name, param in model.named_parameters():
        if name.endswith(".bias"):
            param.data.fill_(0)
        elif name.startswith("layers.0"): # La primera capa no tiene una ReLU aplicada a la entrada
            param.data.normal_(0, 1/math.sqrt(param.shape[1]))
        else:
            param.data.normal_(0, math.sqrt(2)/math.sqrt(param.shape[1]))

model = BaseNetwork(act_fn=nn.ReLU()).to(device)
kaiming_init(model)
```

Inicialización: Kaiming (III)




```
layers.0.weight - Variance: 0.000075
layers.2.weight - Variance: 0.000108
layers.4.weight - Variance: 0.000185
layers.6.weight - Variance: 0.000444
layers.8.weight - Variance: 0.005548
```



Inicialización: Kaiming (IV)

- La varianza se mantiene estable a través de las capas.
- Podemos concluir que la inicialización de Kaiming funciona bien para las redes basadas en ReLU.
- Para Leaky-ReLU, etc., tenemos que ajustar ligeramente el factor de la varianza, ya que la mitad de los valores ya no se establecen en cero.

Optimización del modelo

Algoritmos de optimización

Optimización

- El optimizador es responsable de actualizar los parámetros de la red dados los gradientes.
- Como en, $w^t = f(w^{t-1}, g^t, ...)$.
- Donde, $g^t = \nabla_{w^{(t-1)}} \mathcal{L}^{(t)}$.
- Un parámetro adicional común a esta función es la tasa de aprendizaje, aquí denotada por η.

UNIVERSIDAD ADOLFO IBÁÑEZ

Template para optimización

```
class PlantillaParaOptimizador:
    def __init__(self, params, lr):
        self.params = list(params)
        self.lr = lr
    def zero_grad(self):
        for p in self.params:
            if p.grad is not None:
                p.grad.detach_()
                p.grad.zero_()
    @torch.no_grad()
    def step(self):
        for p in self.params:
            if p.grad is None:
                continue
            self.update_param(p)
    def update_param(self, p):
        raise NotImplementedError
```

Stochastic Gradient Descent (SGD)

$$w^{(t)} = w^{(t-1)} - \eta \cdot g^{(t)}$$

```
class SGD(PlantillaParaOptimizador):

    def __init__(self, params, lr):
        super().__init__(params, lr)

    def update_param(self, p):
        p_update = -self.lr * p.grad
        p.add_(p_update) # Ahorra memoria, no crea grafo computacional
```

Stochastic Gradient Descent con momentum (SGD)

$$m^{(t)} = \beta_1 m^{(t-1)} + (1 - \beta_1) \cdot g^{(t)}$$
$$w^{(t)} = w^{(t-1)} - \eta \cdot m^{(t)}$$

```
1  class SGDMomentum(PlantillaParaOptimizador):
2      def __init__(self, params, lr, momentum=0.0):
3          super().__init__(params, lr)
4          self.momentum = momentum
5          self.param_momentum = {p: torch.zeros_like(p.data) for p in self.params}
6      def update_param(self, p):
7          self.param_momentum[p] = (1 - self.momentum) * p.grad + self.momentum *
               self.param_momentum[p]
8          p_update = -self.lr * self.param_momentum[p]
9          p.add_(p_update)
```

Adam (I)

Adam combina la idea del *momemtum* con la **tasa de aprendizaje adaptativo**. Esta se basa en un promedio exponencial de los gradientes al cuadrado, es decir, la norma de gradientes. Además, agregamos una corrección de sesgo para el *momentum* y la tasa de aprendizaje adaptativo para las primeras iteraciones:

$$
\begin{aligned}
m^{(t)} &= \beta_1 m^{(t-1)} + (1-\beta_1)\cdot g^{(t)} \\
v^{(t)} &= \beta_2 v^{(t-1)} + (1-\beta_2)\cdot \left(g^{(t)}\right)^2 \\
\hat{m}^{(t)} &= \frac{m^{(t)}}{1-\beta_1^t}, \hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t} \\
w^{(t)} &= w^{(t-1)} - \frac{\eta}{\sqrt{v^{(t)}}+\epsilon} \circ \hat{m}^{(t)}
\end{aligned}
$$

UNIVERSIDAD ADOLFO IBÁÑEZ

Adam (II)

```
class Adam(PlantillaParaOptimizador):

    def __init__(self, params, lr, beta1=0.9, beta2=0.999, eps=1e-8):
        super().__init__(params, lr)
        self.beta1 = beta1
        self.beta2 = beta2
        self.eps = eps
        self.param_step = {p: 0 for p in self.params}
        self.param_momentum = {p: torch.zeros_like(p.data) for p in self.params}
        self.param_2nd_momentum = {p: torch.zeros_like(p.data) for p in self.
            params}
```

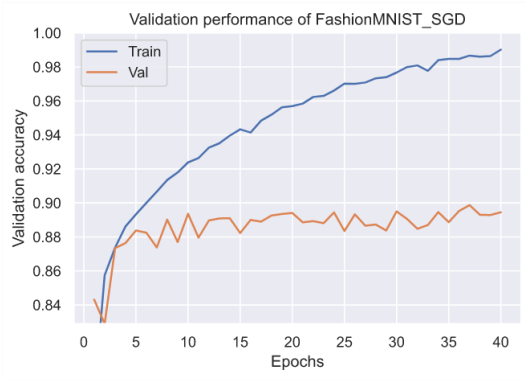
UNIVERSIDAD ADOLFO IBÁÑEZ

Adam (III)

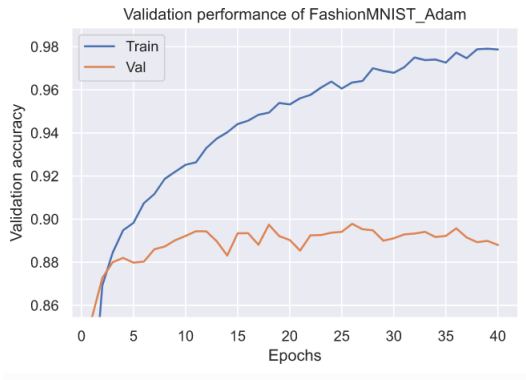
```
class Adam(PlantillaParaOptimizador):
    def update_param(self, p):
        self.param_step[p] += 1

        self.param_momentum[p] = (1 - self.beta1) * p.grad + self.beta1 * self.param_momentum[p]
        self.param_2nd_momentum[p] = (1 - self.beta2) * (p.grad)**2 + self.beta2 * self.param_2nd_momentum[p]

        bias_correction_1 = 1 - self.beta1 ** self.param_step[p]
        bias_correction_2 = 1 - self.beta2 ** self.param_step[p]

        p_2nd_mom = self.param_2nd_momentum[p] / bias_correction_2
        p_mom = self.param_momentum[p] / bias_correction_1
        p_lr = self.lr / (torch.sqrt(p_2nd_mom) + self.eps)
        p_update = -p_lr * p_mom

        p.add_(p_update)
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Comparación de resultados



========== Test accuracy: 89.09% ==========



========== Test accuracy: 89.46% ==========

UNIVERSIDAD ADOLFO IBÁÑEZ

Referencias

- **Welcome to the UvA Deep Learning Tutorials!**. `https://uvadlc-notebooks.readthedocs.io/en/latest/index.html`.
- **Pattern Recognition and Machine Learning.** Christopher M. Bishop. Springer. 2006.
- **The Elements of Statistical Learning: Data Mining, Inference, and Prediction.** Trevor Hastie, Robert Tibshirani, Jerome Friedman. Springer. 2016.
- **Practical Data Science: Deep learning**. J. Zico Kolter. `http://www.datasciencecourse.org/slides/deep_learning.pdf`
- **A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way**. Sumit Saha. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

UNIVERSIDAD ADOLFO IBÁÑEZ

¿Preguntas?