Estrategias para mejorar funciones en la arquitectura de una red neuronal

Tomando tu modelo base (un **MLP de clasificación con `make_moons`**) y pensando en **situaciones en las que el primer experimento con Adam + CrossEntropy + ReLU "se queda corto"**, aquí van **tres combinaciones sugeridas**, cada una pensada para un tipo de dificultad distinta (datos escasos, complejos o desbalanceados).

🎯 1. Datos escasos o poco representativos

El modelo sobreajusta fácilmente, la pérdida baja rápido pero la validación no mejora.

Cambios sugeridos:

Elemento	Reemplazo	Justificación
Optimizador	`SGD + Momentum`	Más estable en datasets pequeños: evita adaptarse en exceso al ruido de cada minibatch.
Función de error	`Huber Loss`	Mezcla de MSE y MAE: penaliza con suavidad los errores pequeños, robusta ante outliers.
Función de activación	`Leaky ReLU`	Evita que las neuronas "mueran" cuando hay pocas muestras informativas.

 Intuición:
Es como decirle al modelo "aprende con cautela y tolerancia". Huber reduce la agresividad del castigo y el momentum le da inercia a los cambios, evitando sobreajuste.

En código (PyTorch Lightning):

```
loss = F.huber_loss(logits, F.one_hot(y, num_classes=2).float())
optimizer = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
activation = nn.LeakyReLU()
```

🧗 2. Datos muy ruidosos o con fronteras complejas

El modelo parece no converger, la pérdida oscila, la superficie del error es irregular.

Cambios sugeridos:

Elemento	Reemplazo	Justificación
Optimizador	`RMSProp`	Diseñado para gradientes inestables (RNNs, datos secuenciales o ruidosos). Adapta el paso a cada dimensión.
Función de error	`Log-Cosh Loss`	Suaviza los gradientes, evita explosiones de error por valores atípicos.
Función de activación	`ELU (Exponential Linear Unit)`	Introduce suavidad en el rango negativo y mejora la estabilidad de la retropropagación.

 Intuición:
Como un pintor que cambia de pincel para suavizar trazos bruscos. El modelo "fluye" mejor por la superficie del error.

En código:

```
loss = torch.mean(torch.log(torch.cosh(logits - F.one_hot(y, num_classes=2).float())))
optimizer = torch.optim.RMSprop(self.parameters(), lr=self.lr)
activation = nn.ELU()
```

⚖️ 3. Datos desbalanceados o clases raras

El modelo ignora clases minoritarias; el accuracy parece alto pero el recall de una clase es bajo.

Cambios sugeridos:

Elemento	Reemplazo	Justificación
Optimizador	`Nesterov Accelerated Gradient (NAG)`	Permite anticipar la dirección del gradiente, mejora la convergencia en problemas difíciles.
Función de error	`Focal Loss`	Pondera más los errores de clases raras y atenúa los de clases frecuentes.
Función de activación	`Swish`	Función suave y autoajustada, ayuda a modelar relaciones no lineales en datos desbalanceados.

 Intuición:
El modelo se vuelve más “atento” a los ejemplos difíciles. Focal Loss concentra el esfuerzo donde el modelo aún falla.

En código:

```
def focal_loss(logits, targets, alpha=0.25, gamma=2.0):
    ce_loss = F.cross_entropy(logits, targets, reduction="none")
    pt = torch.exp(-ce_loss)
    return (alpha * (1 - pt) ** gamma * ce_loss).mean()

optimizer = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
nesterov=True)
activation = lambda: nn.SiLU()  # Swish = SiLU en PyTorch
```

🧠 Resumen comparativo

Contexto problemático	Optimizador sugerido	Función de error	Activación	Efecto esperado
Datos escasos o inestables	SGD + Momentum	Huber	Leaky ReLU	Estabiliza, evita sobreajuste
Datos ruidosos o muy complejos	RMSProp	Log-Cosh	ELU	Suaviza gradientes, mejora convergencia
Datos desbalanceados	NAG	Focal Loss	Swish (SiLU)	Focaliza aprendizaje en clases raras

---

Si quieres, puedo integrarte **estas tres variantes directamente en el modelo base Lightning**, de modo que los estudiantes puedan ejecutarlas con un parámetro `config="escaso"`, `"ruidoso"`, o `"desbalanceado"` y ver en W&B cómo cambia el comportamiento.
¿Quieres que te lo deje implementado así?