PARÁMETROS DE CÓDIGO DE PROYECTO

Nombre		Código	Descripción	Parámetros útiles	Mejor para
Parámetros de optimización de pesos del modelo basado en gradientes para actualización de los pesos de las neuronas	Gradiente Descendiente Estocástico (SGD)	torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)	Método clásico basado en el gradiente	Momentum para acelerar la convergencia	Problemas simples, redes pequeñas, y datos bien normalizados
	Adam (Adaptive Moment Estimation)	torch.optim.Adam(model.parameters(), lr=0.001)	Variante mejorada de SGD que ajusta automáticamente la tasa de aprendizaje		Redes profundas y datos ruidosos
	AdamW (Versión mejorada de Adam con regularización L2 correcta)	torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)	Similar a Adam, pero maneja mejor la regularización (L2)		Modelos grandes como transformers
	RMSprop (Root Mean Square Propagation)	torch.optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)	Divide la tasa de aprendizaje por la raíz cuadrada de promedios pasados		Redes recurrentes (RNN, LSTM)
	Adagrad (Adaptive Gradient Algorithm)	torch.optim.Adagrad(model.parameters(), lr=0.01)	Reduce la tasa de aprendizaje a medida que avanza el entrenamiento		Datos dispersos (NLP, embeddings)
	Adadelta	torch.optim.Adadelta(model.parameters(), rho=0.9)	Similar a Adagrad, pero sin necesidad de definir una tasa de aprendizaje inicial		Datos con escalas muy diferentes
	Adamax (Versión de Adam basada en norma infinita)	torch.optim.Adamax(model.parameters(), lr=0.002)	Más estable que Adam en ciertas condiciones		Datos grandes y redes muy profundas
	LBFGS (Limited-memory BFGS)	torch.optim.LBFGS(model.parameters(), lr=0.01)	Método de optimización de segundo orden (más costoso pero preciso)		Pequeños conjuntos de datos con funciones de pérdida suaves
Funciones de pérdida para regresión. Estas funciones se usan cuando la salida es un valor numérico	Error Cuadrático Medio (MSE - Mean Squared Error)	torch.nn.MSELoss()	Penaliza errores grandes de manera cuadrática		Bueno para regresión en general
	Error Absoluto Medio (MAE - Mean Absolute Error / L1 Loss)	torch.nn.L1Loss()	Penaliza errores de forma lineal en lugar de cuadrática		Más robusto a valores atípicos (outliers)
	Error Huber (Huber Loss - mezcla entre MSE y MAE)	torch.nn.SmoothL1Loss()	Usa MSE para errores pequeños y MAE para errores grandes		Bueno cuando hay outliers en los datos
	Log-Cosh Loss (No está en PyTorch, pero se puede implementar)	def log_cosh_loss(y_pred, y_true): return torch.mean(torch.log(torch.cosh(y_pred - y_true)))	Similar a Huber pero más suave		
Para clasificación (salida categórica). Estas funciones se usan cuando la salida es una categoría o clase	Entropía Cruzada (Cross Entropy Loss) - Para clasificación multiclase	torch.nn.CrossEntropyLoss()	Para clasificación con múltiples clases (ej. softmax)	Ideal para redes que clasifican imágenes o texto	Importante: No necesita softmax en la salida, ya que lo aplica internamente
	Binary Cross Entropy (BCE) - Para clasificación binaria	torch.nn.BCELoss()	Se usa con una salida que pase por sigmoid		Ideal para problemas de sí/no, 0/1, positivo/negativo
	Binary Cross Entropy con Logits (BCEWithLogitsLoss)	torch.nn.BCEWithLogitsLoss()	Similar a BCE, pero más estable porque aplica sigmoid internamente		Mejor opción que BCELoss() si no aplicaste sigmoid en la salida
	KL Divergence (Kullback-Leibler Loss)	torch.nn.KLDivLoss()	Mide la diferencia entre dos distribuciones de probabilidad		Útil en redes con softmax, modelos de compresión y aprendizaje bayesiano
Para segmentación y detección de objetos. Funciones usadas en visión por computadora y procesamiento de imágenes	Dice Loss (No está en PyTorch, pero se puede implementar fácilmente)	def dice_loss(y_pred, y_true): smooth = 1.0 intersection = torch.sum(y_pred * y_true) return 1 - (2.*intersection+smooth)/(torch.sum(y_pred)+torch.sum(y_true)+smooth)			Muy usado en segmentación de imágenes.
	IoU Loss (Intersection over Union)				Similar a Dice Loss, pero mide la superposición entre predicción y verdad

	Focal Loss (Para datos desbalanceados en clasificación)	class FocalLoss(nn.Module):
		def __init__(self, gamma=2):
		super(FocalLoss, self).__init__()
		self.gamma = gamma
		def forward(self, inputs, targets):
					Útil en detección de objetos (cuando hay muchas clases desbalanceadas)

		BCE_loss = torch.nn.functional.binary_cross_entropy(inputs, targets, reduction='none') pt = torch.exp(-BCE_loss) focal_loss = (1 - pt) ** self.gamma * BCE_loss return focal_loss.mean()			
Adam / AdamW → La opción más usada en Deep Learning. **SGD con momentum → Bueno para visión por computadora y entrenamiento estable.** **RMSprop → Ideal para RNN / LSTM.** **Adagrad / Adadelta → Para datos dispersos (ej. texto).** **LBFGS → Para problemas donde necesitas alta precisión en redes pequeñas.**					
Regresión: MSELoss() o SmoothL1Loss() si hay outliers. **Clasificación binaria: BCEWithLogitsLoss() (mejor que BCELoss()).** **Clasificación multiclase: CrossEntropyLoss().** **Segmentación de imágenes: Dice Loss o IoU Loss.** **Datos desbalanceados: Focal Loss.**					