Aula 6: Regularização — Dropout e Early Stopping

Aléssio Miranda Júnior

Agosto - 2026/2

O que você verá neste capítulo

Na aula anterior, diagnosticamos o Overfitting como a doença mais perigosa de qualquer sistema de Machine Learning. Neste capítulo, apresentaremos as "vacinas" contra o sobreajuste: as **Técnicas de Regularização**. Começaremos pela regularização L1/L2 (penalização nos pesos), avançaremos para a técnica do *Dropout* de Srivastava et al. (2014), e fecharemos com o *Early Stopping* e o conceito de *Callbacks* do Keras, ferramentas que automatizam a detecção e a interrupção do treinamento no momento exato em que a memorização se instala.

1 O Antídoto para o Overfitting

Na aula anterior, diagnosticamos o Overfitting como a armadilha mais perigosa do Machine Learning: uma rede que "decora" os dados de treino e fracassa diante de qualquer exemplo inédito. A pergunta que fica é: *como curar essa doença sem matar o paciente?*

1.1 A Analogia do Personal Trainer

Imagine que você decidiu treinar para uma maratona. No começo, corre todos os dias com entusiasmo, mas sem orientação profissional. Nas primeiras semanas, o progresso é evidente — você corre mais rápido e mais longe. Porém, sem controle, o corpo começa a dar sinais de exaustão: lesões musculares, dores articulares e, eventualmente, uma fratura por estresse. Você treinou **demais** (*overfitting corporal*), e o corpo "memorizou" vícios de postura que agora o impedem de competir.

Um personal trainer resolve exatamente esse problema. Ele não impede que você treine — impede que você **se destrua treinando**. Ele limita a carga máxima dos pesos na academia (*Regularização L2*), alterna aleatoriamente

os grupos musculares trabalhados para evitar dependência excessiva de um único músculo (*Dropout*), e monitora seus sinais vitais para interromper o treino no momento exato em que o corpo começar a regredir (*Early Stopping*).

As **Técnicas de Regularização** são, para a rede neural, exatamente o que o personal trainer é para o atleta: métodos que forçam o modelo a aprender de maneira mais robusta, penalizando a complexidade excessiva sem sacrificar a capacidade de representação. A solução nunca é simplificar a rede ao ponto de torná-la inútil (Underfitting) — é discipliná-la.

◇ Informação: Conexão com IA I

Em IA I, vocês viram que os **Algoritmos Genéticos** utilizam operadores de **mutação** para injetar diversidade e impedir convergência prematura da população. O Dropout que estudaremos neste capítulo faz algo conceitualmente análogo: injeta aleatoriedade na rede para impedir que os neurônios "convirjam" para uma solução viciada. A regularização, assim como a mutação genética, é uma forma de **diversificação forçada**.

1.2 Regularização L1 e L2 (Penalização nos Pesos)

A abordagem clássica é adicionar um termo de penalidade diretamente à Função de Perda. Ao invés de minimizar apenas o erro puro, o otimizador agora precisa minimizar o erro **mais** o custo de manter pesos grandes:

$$\mathcal{L}_{total} = \mathcal{L}_{dados} + \lambda \cdot \Omega(\mathbf{W}) \tag{1}$$

Onde λ é o hiperparâmetro de regularização (que controla a intensidade da penalidade) e $\Omega(\mathbf{W})$ é a penalidade sobre os pesos:

- **L2 (Ridge):** $\Omega = \sum w_i^2$ — Penaliza pesos grandes, forçando-os a valores menores e mais distribuídos. É a técnica mais comum em Deep Learning.
- **L1 (Lasso):** $\Omega = \sum |w_i|$ — Tende a zerar pesos irrelevantes completamente, criando um modelo mais esparso.

• Teoria: Intuição Geométrica da Regularização L2

A regularização L2 pode ser entendida geometricamente: ela restringe os pesos a uma "esfera" em torno da origem no espaço de parâmetros. Sem regularização, os pesos podem crescer arbitrariamente para memorizar padrões de ruído. Com L2, o otimizador é forçado a encontrar soluções dentro dessa esfera, preferindo pesos pequenos e difusos — o que resulta em fronteiras de decisão mais suaves e generalizáveis.

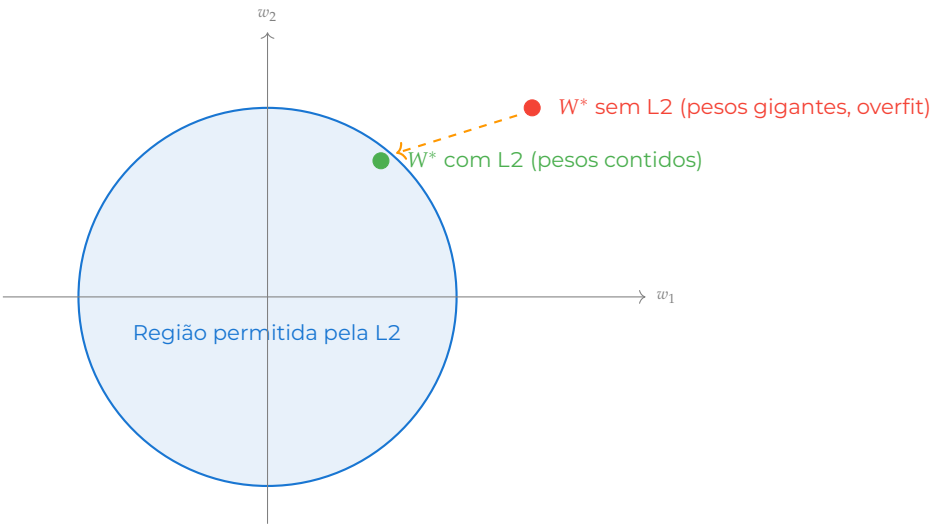


Figura 1: A regularização L2 restringe os pesos a uma esfera (região azul). O ótimo migra de pesos gigantes (vermelho, overfit) para pesos contidos (verde, generalizável).

▷ Exemplo: Exemplo Numérico: L2 em Ação

Considere uma rede com apenas 3 pesos: $W = [4.0,\ 0.1,\ 3.5]$. Suponha que a loss pura dos dados seja $\mathcal{L}_{dados} = 0.50$ e que usemos $\lambda = 0.01$.

Sem regularização:

$$\mathcal{L}_{total} = 0.50$$

Com L2 ($\lambda = 0.01$):

$$\Omega = w_1^2 + w_2^2 + w_3^2 = 4.0^2 + 0.1^2 + 3.5^2 = 16.0 + 0.01 + 12.25 = 28.26$$

$$\mathcal{L}_{total} = 0.50 + 0.01 \times 28.26 = 0.50 + 0.2826 = \mathbf{0.7826}$$

Observe: os pesos grandes ($w_1 = 4.0$ e $w_3 = 3.5$) são os que mais "pagam" na penalidade quadrática. O peso pequeno ($w_2 = 0.1$) contribui com apenas 0.01 à penalidade. Isso pressiona o otimizador a **redistribuir** a informação entre todos os pesos em vez de concentrá-la em poucos — exatamente a mesma lógica de um portfólio financeiro diversificado que vocês viram em Engenharia Econômica.

No Keras, a regularização L2 é aplicada diretamente na declaração da camada:

```
from tensorflow.keras.regularizers import l2

model.add(Dense(64, activation='relu',
                kernel_regularizer=l2(0.01)))
```

2 A Técnica do Dropout

O *Dropout* (Srivastava et al., 2014) é uma das técnicas de regularização mais simples e, paradoxalmente, uma das mais eficientes já inventadas na história do Deep Learning.

Durante o treinamento, a cada época, a camada de *Dropout* "desliga" aleatoriamente uma porcentagem dos neurônios (por exemplo, 20%). Isso significa que esses neurônios adormecidos não farão cálculos e não terão seus pesos ajustados naquele ciclo específico. Na época seguinte, um conjunto diferente de neurônios é desligado. O processo é estocástico e imprevisível.

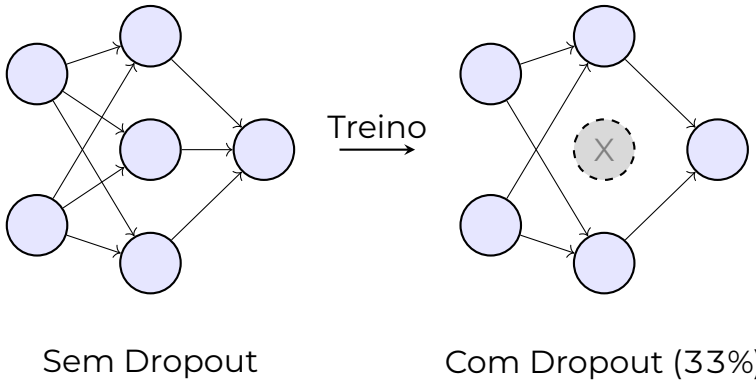


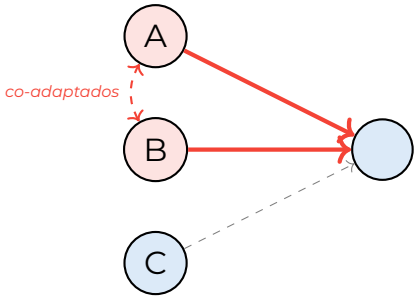
Figura 2: Ilustração da ação da camada Dropout, desligando aleatoriamente neurônios durante o treinamento. Na inferência, todos os neurônios são reativados com pesos escalados.

2.1 Por Que o Dropout Funciona? A Analogia do Seminário Universitário

Por que isso funciona? Imagine que o professor de uma disciplina designou um seminário em grupo para 5 alunos. Se o professor **sempre** mantiver os mesmos 5 alunos juntos, rapidamente dois ou três alunos "estrelas" dominarão o trabalho enquanto os demais ficam passivos — a clássica *panelinha* acadêmica. O grupo "decora" uma divisão de trabalho viciada.

Agora imagine que, a cada semana, o professor **sorteia aleatoriamente** quais alunos serão excluídos da apresentação. Nenhum aluno sabe se estará presente na próxima rodada. O resultado? **Todos** são obrigados a entender **todo** o conteúdo, porque qualquer um pode ser chamado a apresentar qualquer parte. A panelinha é destruída pela aleatoriedade.

Essa é exatamente a mecânica do Dropout. Como os neurônios nunca sabem se seus "colegas" estarão ligados no próximo ciclo, eles não podem criar codependências. Cada neurônio é forçado a aprender aspectos mais úteis e independentes do dado, tornando a rede inteira mais resiliente.



Sem Dropout: A e B "combinam" para memorizar. C fica inútil.

Figura 3: O problema da co-adaptação: sem Dropout, neurônios A e B criam dependências mútuas para memorizar padrões específicos. O Dropout quebra essa dinâmica ao desligar neurônios aleatoriamente.

Do ponto de vista matemático, o Dropout cria implicitamente um *ensemble* (comitê) de 2^n sub-redes distintas, onde n é o número de neurônios. Na hora da inferência, ativar todos os neurônios é equivalente a calcular a média das predições desse enorme comitê. Os pesos são automaticamente escalados por $(1 - p)$ para compensar o fato de que, durante o treino, apenas uma fração dos neurônios estava ativa.

△ Importante: Dropout: Treino vs. Inferência

Um detalhe crucial: o Dropout **só age durante o treinamento**. Na hora de fazer previsões em produção (inferência), **todos** os neurônios são reativados. O Keras cuida disso automaticamente — quando você chama `model.predict()`, o Dropout é desligado. Se alguém esquecesse de desativá-lo na inferência, as previsões seriam ruidosas e inconsistentes, pois a cada chamada um subconjunto diferente de neurônios estaria ativo.

▶ Prática: Dropout no Keras

No Keras, o Dropout é uma camada inserida entre as camadas densas:

```
from tensorflow.keras.layers import Dense, Dropout

model = Sequential()
model.add(Dense(128, activation='relu', input_shape=(10,)))
model.add(Dropout(0.3))  # Desliga 30% dos neuronios
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))  # Desliga 20% dos neuronios
model.add(Dense(1, activation='sigmoid'))
```

Regra Prática: Taxas de Dropout entre 0.2 e 0.5 são as mais comuns na literatura. Valores acima de 0.5 podem causar Underfitting (desligar metade dos neurônios é agressivo demais para a maioria dos problemas).

3 Early Stopping (Parada Antecipada e Callbacks)

Vimos na aula anterior que o Overfitting se manifesta visualmente através da divergência de trajetórias: a perda de treino (`loss`) continua decrescendo monotonamente em direção a zero, enquanto a perda no conjunto de validação (`val_loss`) atinge um ponto de mínimo e passa a se degradar.

No entanto, pausar o treinamento de uma rede neural no momento exato desse vale não é uma tarefa trivial. Em ambientes de produção e pesquisa, modelos treinam durante horas, dias ou semanas. Não é viável (e nem metodologicamente rigoroso) depender de um engenheiro observando telas de terminal para interromper o processo via comando manual (*Ctrl+C*). Para transformar essa inspeção em um processo determinístico e automatizado, utilizamos o conceito de **Callbacks**, com destaque absoluto para o mecanismo de **Early Stopping** (Parada Antecipada).

3.1 A Arquitetura de um Callback: O Vigia da Rede

Em engenharia de software, um *callback* é uma função ou objeto passado como argumento que é executado em resposta a eventos específicos do ciclo de vida da aplicação. No Keras, os callbacks são sentinelas que interceptam o fluxo do treinamento em marcos chave: no início de cada época (`on_epoch_begin`), no término de cada lote de dados (`on_batch_end`) ou ao final de cada época completa (`on_epoch_end`).

O `EarlyStopping` é um callback que atua no evento `on_epoch_end`. Sua responsabilidade é inspecionar uma métrica de validação definida pelo engenheiro (por padrão, `val_loss` para minimização, ou `val_accuracy` para maximização) e aplicar um protocolo rigoroso de decisão antes de autorizar a execução da época seguinte.

3.2 Os Três Pilares da Tomada de Decisão

Para operar de forma robusta em cenários reais — onde o gradiente estocástico introduz ruído e flutuações nas curvas de aprendizado — o Early Stopping é parametrizado por três mecanismos fundamentais:

3.2.1 1. A Paciência (*Patience*) e a Dinâmica de Reset

O treinamento por descida de gradiente em mini-batches raramente produz curvas perfeitamente monotônicas. É natural que o `val_loss` sofra pequenas oscilações de subida temporária decorrentes da aleatoriedade dos lotes, antes de retomar a trajetória descendente. Se o algoritmo interrompesse o treino no primeiro milésimo de segundo em que a métrica subisse, estaríamos diante de um **falso alarme**:

- **Paciência Muito Curta ($p = 1, 2$):** A rede é interrompida prematuramente devido ao ruído estocástico intrínseco do lote, antes de consolidar sua capacidade de representação.
- **Paciência Excessivamente Longa ($p = 100, 200$):** A rede permanece rodando dezenas de épocas após ter entrado em franco sobreajuste, desperdiçando tempo de desenvolvimento e recursos computacionais de GPU.

O hiperparâmetro `patience=p` estabelece uma contagem regressiva de tolerância. A cada época em que o modelo não atinge um novo recorde histórico de menor perda, o contador de estagnação avança $(1/p, 2/p, \dots)$.

O Mecanismo de Reset: Suponha que o modelo atinja uma perda de $0{,}85$ na época 12 e permaneça oscilando em torno de $0{,}90$ até a época 16 (4 épocas de estagnação). Se tivermos configurado patience $= 4$, o algoritmo abortaria na época 16. Contudo, com patience $= 5$, o treino atinge a época 17, onde a perda cai para $0{,}81$. Como essa marca representa um novo mínimo global histórico, o Callback executa imediatamente o **RESET do contador para** $0/5$. A rede ganha novo fôlego e o ciclo recomeça a partir do novo patamar.

3.2.2 2. A Tolerância (`min_delta`) contra o Falso Progresso

Quando a rede se aproxima do fundo da bacia de atração do erro, o módulo do gradiente decresce acentuadamente. Nessa região de platô, a perda de validação pode continuar caindo em passos microscópicos (por exemplo, de $0{,}8200$ para $0{,}8199$, um decréscimo de $\Delta = 0{,}0001$).

Sem um critério de corte, esse ganho ínfimo é interpretado como uma "vitória", acionando o reset da paciência e forçando a GPU a processar centenas de épocas adicionais que não agregam valor estatístico perceptível ao poder preditivo do modelo.

O hiperparâmetro `min_delta` define o limiar mínimo de melhoria considerado significativo:

$$\mathcal{L}_{val}^{(novo)} < \mathcal{L}_{val}^{(min)} - \text{min\_delta} \tag{2}$$

Se o decréscimo for inferior a `min_delta`, o Callback ignora a variação numérica e **continua avançando a contagem de paciência**, protegendo o projeto contra custos computacionais desnecessários.

3.2.3 3. O Disparo do STOP e o Rebobinamento de Pesos (`restore_best_weights`)

O Early Stopping é um mecanismo fundamentalmente retrospectivo: ele só possui certeza matemática de que o ponto ótimo passou após observar p épocas consecutivas de estagnação ou subida contínua da perda.

Isso cria uma consequência direta: no momento em que a paciência se esgota e a instrução de STOP é disparada (época $N = t_{\text{ótimo}} + p$), os pesos que residem na memória da rede já foram corrompidos por p épocas de sobreajuste.

△ Importante: O Erro Mais Comum: Omitir `restore_best_weights`

Se o desenvolvedor instanciar o Early Stopping sem o parâmetro `restore_best_weights=True`, o modelo final entregue será o da época de parada (época N), com métricas de generalização severamente degradadas pelo overfitting.
Com `restore_best_weights=True`, o Callback salva silenciosamente um *checkpoint* dos tensores de peso na memória RAM a cada novo recorde da métrica. No instante em que o STOP é acionado, o algoritmo descarta os tensores degradados da época N e **rebobina** a arquitetura para os parâmetros exatos do vale ótimo histórico.

3.3 Anatomia Gráfica da Decisão

A Figura 4 sintetiza a interação sinérgica entre a paciência, o mecanismo de reset, o filtro de tolerância `min_delta` e o rebobinamento de pesos para o estado ótimo de generalização.

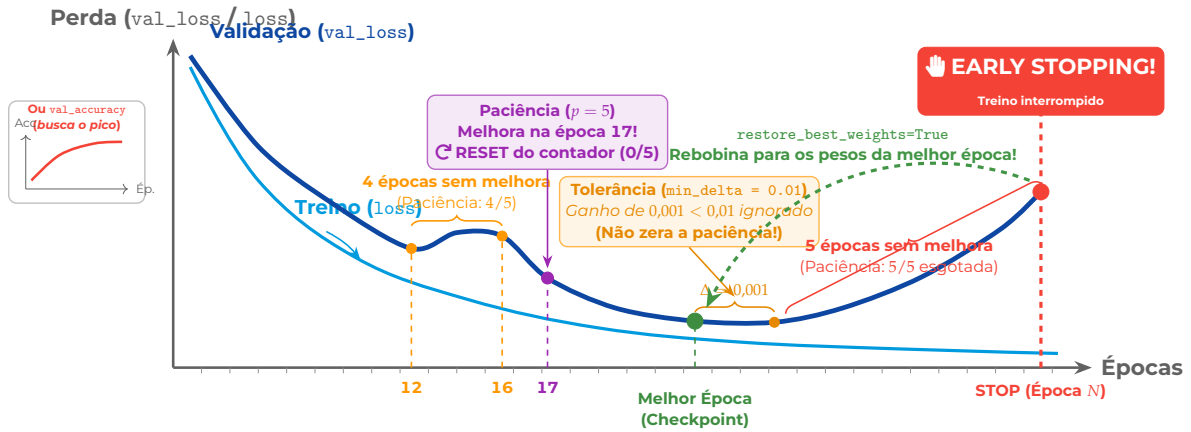


Figura 4: Anatomia completa do Early Stopping: na Região 1, a paciência sofre reset após um novo recorde na época 17. Na Região 2, melhorias microscópicas abaixo de `min_delta` são ignoradas. Na Região 3, o esgotamento da paciência dispara o STOP e os pesos ótimos do vale são restaurados.

3.4 Implementação Prática no Keras

No Keras, a configuração do sentinela é implementada pela classe `EarlyStopping`, passada na lista do argumento `callbacks` durante a chamada do método `fit()`:

```
from tensorflow.keras.callbacks import EarlyStopping

# 1. Instanciacao do Sentinela
early_stopping = EarlyStopping(
    monitor='val_loss',          # Metrica vigiada no conjunto de validacao
    patience=10,                 # Tolera ate 10 epocas consecutivas de estagnacao
    min_delta=0.001,             # Ganho minimo para qualificar progresso real
    mode='min',                  # 'min' para loss; 'max' para acuracia
    restore_best_weights=True,   # Restaura os tensores do melhor checkpoint
    verbose=1                    # Notifica no console o encerramento do treino
)

# 2. Execucao com teto de seguranca elevado
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=500,                  # Teto de seguranca (o callback decide o fim)
    batch_size=32,
    callbacks=[early_stopping]
)
```

◇ **Informação: Por que Definir `epochs=500`?**

Com o Early Stopping ativo, o parâmetro `epochs=500` deixa de ser uma meta a ser atingida e passa a atuar apenas como um **teto máximo de segurança**. O engenheiro não precisa mais adivinhar qual o número ideal de épocas para convergir sem sobreajustar: basta definir um número deliberadamente alto e delegar a decisão ótima de interrupção ao Callback.

4 Otimizadores Modernos: Adam e RMSprop

Nas primeiras aulas, simulamos o *Gradient Descent* padrão (SGD) usando uma taxa de aprendizado fixa (α). Contudo, terrenos matemáticos reais são acidentados e heterogêneos: em certas regiões a curvatura é íngreme e exige passos curtos, em outras o declive é suave e pede aceleração.

Vocês viram em **Cálculo III** que superfícies multivariáveis possuem curvaturas diferentes em cada direção — a mesma taxa de passo que funciona no eixo x pode ser catastrófica no eixo y. O SGD ignora essa heterogeneidade e aplica o mesmo α para todas as dimensões. Os otimizadores modernos resolvem esse problema.

O Keras já traz otimizadores inteligentes que ajustam a taxa de aprendizado dinamicamente **para cada peso individual** durante o treinamento:

- **RMSprop** (Hinton, 2012): Normaliza o gradiente pela média móvel dos quadrados dos gradientes anteriores. Funciona muito bem para dados sequenciais e recorrentes.
- **Adam** (Kingma & Ba, 2015): Combina as vantagens do *Momentum* (que acumula velocidade na direção dominante) com o *RMSprop*. É considerado o **otimizador padrão** da indústria moderna de Deep Learning por sua robustez e convergência rápida.

• Teoria: A Intuição do Adam

O **Adam** mantém duas "memórias" adaptativas para cada peso:

1. **Primeiro Momento (m):** A média móvel dos gradientes. Funciona como um *momentum* que dá inércia à direção de descida, impedindo que o otimizador fique oscilando entre declives opostos.
2. **Segundo Momento (v):** A média móvel dos gradientes ao quadrado. Estima a curvatura local da superfície de erro. Em regiões onde o gradiente é alto e instável, o Adam reduz automaticamente o tamanho do passo para evitar divergência.

A regra de atualização combina ambos:

$$W_{novo} = W_{atual} - \alpha \cdot \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{3}$$

Onde ϵ é uma constante minúscula (10^{-8}) que impede divisão por zero.

A diferença comportamental entre o SGD e o Adam pode ser visualizada como dois caminhantes descendo a mesma montanha:

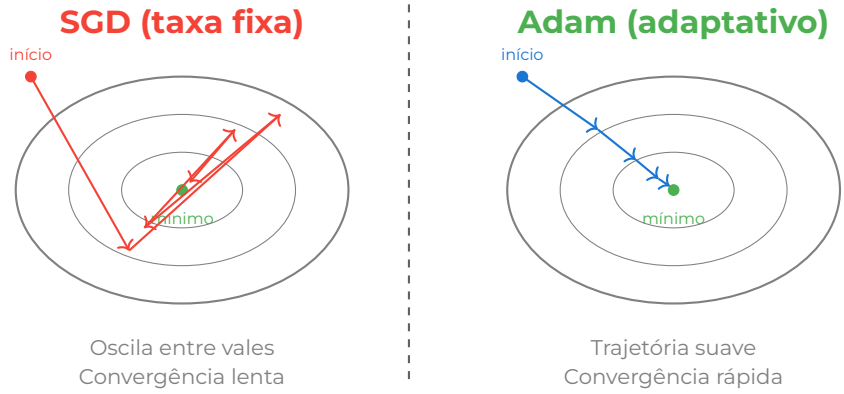


Figura 5: Comparação visual das trajetórias de otimização: o SGD (esquerda) oscila violentamente em direções de alta curvatura, desperdiçando iterações. O Adam (direita) adapta o tamanho do passo por dimensão, convergindo suavemente ao mínimo.

Na compilação do modelo, basta substituir o otimizador:

```
# Ao inves de:
model.compile(optimizer='sgd', ...)

# Usamos:
model.compile(optimizer='adam', ...)
# ou com taxa de aprendizado customizada:
from tensorflow.keras.optimizers import Adam
model.compile(optimizer=Adam(learning_rate=0.001), ...)
```

5 O Arsenal Completo em Ação

Com a combinação do particionamento rigoroso de dados (Treino/Validação/Teste), a penalização L2 nos pesos, o *Dropout* como regularizador estocástico e o *Early Stopping* como sentinela automática, nós finalmente montamos a **caixa de ferramentas profissional e blindada** do Engenheiro de IA. Cada técnica ataca o Overfitting por um ângulo diferente, e a combinação é sinérgica:

Técnica	Analogia	Mecanismo
L2	Limite de carga na academia	Penaliza pesos grandes na loss
Dropout	Rodízio de jogadores	Desliga neurônios aleatoriamente
Early Stopping	Monitor cardíaco	Para o treino ao sinal de regressão
Adam	GPS adaptativo	Ajusta α por peso e por direção

Tabela 1: Síntese das técnicas de regularização e otimização vistas neste capítulo, com suas analogias e mecanismos.

O modelo resultante da aplicação dessas técnicas não apenas aprende melhor, como se adapta impecavelmente ao uso em produção. A diferença prática é visível nos gráficos de treinamento:

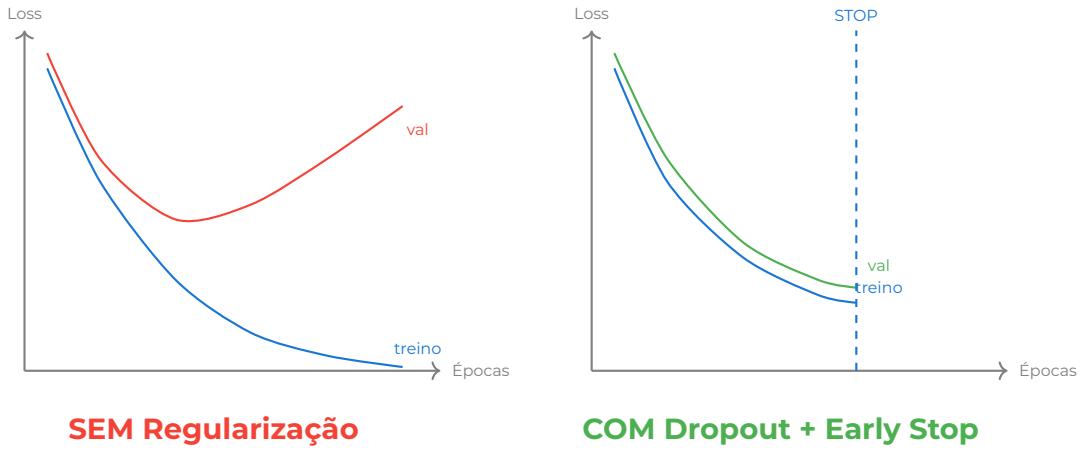


Figura 6: Comparação visual: sem regularização (esquerda), as curvas de treino e validação divergem (Overfitting). Com Dropout + Early Stopping (direita), as curvas caminham juntas e o treinamento para automaticamente no ponto ótimo.

6 Conclusão

Retornemos à analogia do personal trainer com a qual abrimos este capítulo. Sem orientação, o atleta treina até se machucar (Overfitting). Com L2, limitamos a carga máxima. Com Dropout, alternamos os exercícios para evitar vícios posturais. Com Early Stopping, monitoramos os sinais vitais e paramos no momento ideal. E com o Adam, ajustamos o ritmo de treino a cada exercício, respeitando a curvatura do terreno.

O resultado é um modelo **profissional**: que sabe o que aprendeu, não decora o que não entendeu, e está pronto para enfrentar o mundo real. Na prática, a receita que utilizaremos daqui para frente em praticamente todos os projetos da disciplina será:

```
model.compile(optimizer='adam', loss='...', metrics=[...])
model.fit(X_train, y_train,
          validation_data=(X_val, y_val),
          epochs=200,
          callbacks=[EarlyStopping(patience=5,
                     restore_best_weights=True)])
```

Na próxima aula, daremos o salto para além dos dados tabulares e exploraremos como as Redes Neurais processam **imagens** através das Redes Convolucionais (CNNs) e do paradigma revolucionário do **Transfer Learning**. Se até agora trabalhamos com planilhas CSV onde cada coluna é uma *feature* numérica, preparem-se: a entrada será uma **imagem inteira** — e isso muda tudo.

✔ Takeaways

- A **Regularização L2** adiciona $\lambda \sum w_i^2$ à loss, forçando pesos menores, difusos e fronteiras de decisão mais suaves.
- O **Dropout** desliga aleatoriamente neurônios durante o treino, impedindo co-adaptação e criando um ensemble implícito de 2^n sub-redes.
- O **Early Stopping** monitora o `val_loss` e opera com um contador de **paciência** que sofre **reset imediato** a cada novo recorde global histórico.
- O parâmetro `min_delta` define a tolerância mínima de ganho, evitando falsos resets causados por melhorias microscópicas no platô do fundo do vale.
- O parâmetro `restore_best_weights=True` é vital para rebobinar a rede aos pesos do melhor momento histórico, descartando as épocas de sobreajuste final.
- O **Adam** é o otimizador padrão da indústria, combinando Momentum e RMSprop para ajustar α individualmente por peso e por direção.
- A combinação de Normalização + L2 + Dropout + Early Stopping forma o **arsenal profissional completo** contra Overfitting.

✎ Questões: Autoavaliação

1 **[Direta]** Explique por que o Dropout desativa neurônios **apenas durante o treino** e não durante a inferência. O que aconteceria se o Dropout ficasse ativo na hora de fazer previsões em produção?

2 **[Direta]** Durante o treinamento com `EarlyStopping(patience=5)`, a perda de validação estagna por 4 épocas e, na 5ª época, atinge um novo valor mínimo histórico. O que acontece com o contador de paciência da rede?

3 **[Direta]** Qual o perigo prático de omitir o parâmetro `min_delta` em modelos que atingem platôs planos, e por que `restore_best_weights=True` é indispensável para evitar que a rede seja entregue com sobreajuste?

4 **[Reflexiva]** Avalie o *trade-off* na escolha do valor de `patience`: quais são os riscos técnicos de valores muito baixos (ex: $p = 1$ ou 2) versus valores excessivamente altos (ex: $p = 100$)?

5 **[Reflexiva]** Se o Dropout cria um comitê implícito de 2^n sub-redes e o Early Stopping encontra o ponto ótimo na curva de validação, qual é o efeito sinérgico dessas duas técnicas na capacidade de generalização de uma rede profunda?

Lab. 06: Regularização e Dropout

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

⋆ Objetivo do Laboratório

No Lab 05, você provocou Overfitting deliberadamente e viu a "boca de jacaré" entre as curvas de treino e validação. Hoje você vai **curar** aquele modelo doente. Usando o **mesmo dataset**, aplicará três técnicas profissionais — `Dropout`, `Regularização L2` e `EarlyStopping` — e comparará visualmente os gráficos **Antes vs Depois**. Ao final, terá dominado a **caixa de ferramentas completa** contra o Overfitting, a mesma usada em produção por engenheiros do Google e da Meta.

7 Parte 1: O Problema F — Estilo Maratona

Problema F: A Clínica do Modelo Doente

Contexto: No Lab 05, o modelo da *SmartFactory* atingiu 99% de acurácia no treino, mas apenas ≈75% na validação — ele "decorou" os dados em vez de aprender padrões. O CTO da empresa está furioso: *"Gastamos milhares em GPU para um modelo que não funciona em produção!"*. Sua missão é **curar** o modelo usando a caixa de ferramentas de regularização.

Condições do Experimento:

- **Dataset:** Idêntico ao Lab 05 (500 amostras, 10 features, `flip_y=0.1`)
- **Modelo doente (controle):** Rede 256-128-64 sem proteção, 300 épocas
- **Modelo curado (tratamento):** Rede 128-64-32 com Dropout + L2 + Early Stopping

Modelo de Saída: Gráfico comparativo **Antes vs Depois** mostrando:

- **Antes:** Curvas divergentes (boca de jacaré) — Overfitting
- **Depois:** Curvas paralelas (caminham juntas) — Modelo saudável

Critérios de Aprovação:

1. **Early Stopping** ativou antes de 500 épocas
2. **Gap** (loss validação − loss treino) do modelo regularizado < gap do modelo sem proteção
3. Previsões do modelo no intervalo válido $[0, 1]$
4. `restore_best_weights = True` configurado no callback

8 Parte 2: Recriando o Dataset do Lab 05 (5 min)

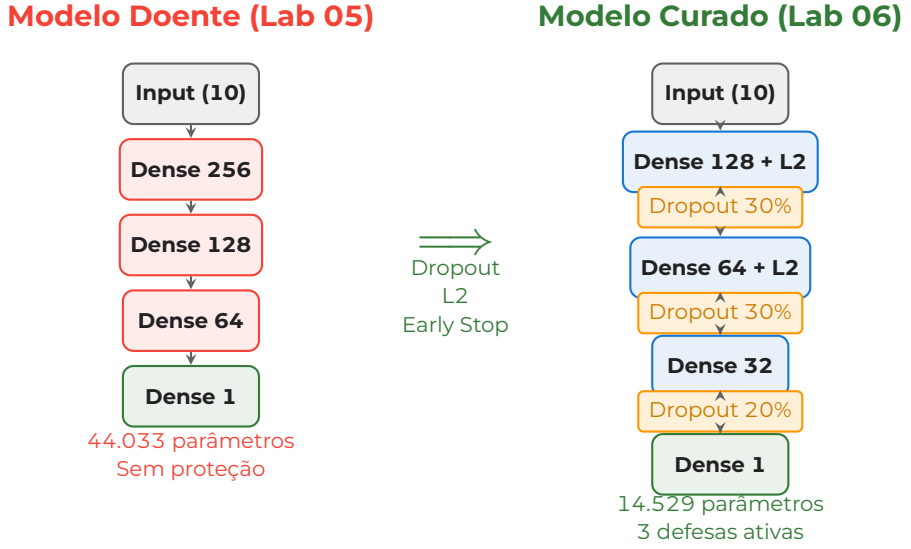
Crie o arquivo `src/modelo_regularizado.py`. Reutilizaremos o mesmo dataset do Lab 05 para garantir comparação justa:

```
import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

import matplotlib
matplotlib.use('Agg')
```

```python
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2

# Mesmo dataset do Lab 05 (reprodutibilidade garantida)
X, y = make_classification(
    n_samples=500, n_features=10,
    n_informative=5, n_redundant=3,
    random_state=42, flip_y=0.1
)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)
```

9 Parte 3: Diagnóstico Visual — Rede Doente vs Curada

Antes de codar, observe o diagrama que compara as duas arquiteturas. A chave está nos blocos de Dropout (representados como “filtros” que desligam neurônios) e na penalidade L2 nos pesos:



10 Parte 4: O Modelo Curado com Dropout + L2 (10 min)

Construa o modelo regularizado. Observe as três camadas de defesa:

```python
def criar_modelo_regularizado():
    """Rede com Dropout + L2 como vacina contra Overfitting."""
    model = Sequential()
    # L2 penaliza pesos grandes na loss
    model.add(Dense(128, activation='relu', input_shape=(10,),
                    kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))    # Desliga 30% dos neurônios

    model.add(Dense(64, activation='relu',
                    kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))

    model.add(Dense(32, activation='relu'))
    model.add(Dropout(0.2))    # Menos agressivo perto da saída

    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
```

• Teoria: As Três Defesas Combinadas

Técnica	Mecanismo	Analogia
Dropout (0.3)	Desliga 30% dos neurônios a cada batch	"Rodízio de funcionários"
L2 (λ=0.001)	Penaliza $\sum w_i^2$ na loss	"Imposto sobre pesos grandes"
Early Stopping	Para o treino quando val_loss sobe	"Sentinela automática"

Nenhuma técnica sozinha resolve todos os casos. Na prática, **combinamos as três** — assim como um médico prescreve múltiplos tratamentos para uma doença resistente.

11 Parte 5: Configurando o Early Stopping (8 min)

Configure o callback que monitora `val_loss` e para o treino automaticamente:

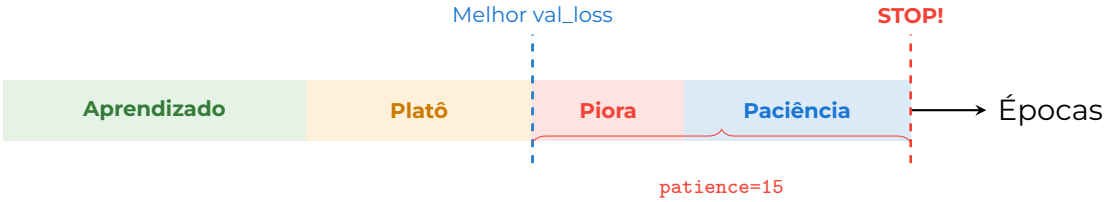
```python
# O "vigia" que monitora a validação
```

```
vigilante = EarlyStopping(
    monitor='val_loss',          # Métrica monitorada
    patience=15,                 # Tolerância: 15 épocas sem melhora
    restore_best_weights=True    # Restaura os pesos do vale!
)
```

△ Importante: O Erro de Esquecer restore_best_weights

Sem `restore_best_weights=True`, o Keras para o treino mas mantém os **últimos** pesos (que já estão contaminados pelo início do Overfitting). Com a flag ativa, ele "viaja no tempo" e restaura os pesos do momento ótimo — a época onde `val_loss` atingiu o mínimo global.

A. Linha do Tempo do Early Stopping



12 Parte 6: Treinamento Comparativo (12 min)

Treine ambos os modelos (doente e curado) com o mesmo dataset para comparação justa:

```
# === MODELO CURADO (com proteção) ===
modelo_reg = criar_modelo_regularizado()
hist_reg = modelo_reg.fit(
    X_train_norm, y_train,
    epochs=500,              # Teto alto (Early Stopping decide)
    batch_size=16,
    validation_split=0.2,
    callbacks=[vigilante],
    verbose=0
)

# === MODELO DOENTE (controle, sem proteção) ===
def criar_modelo_sem_protecao():
    """Rede do Lab 05 sem nenhuma defesa."""
    model = Sequential()
    model.add(Dense(256, activation='relu', input_shape=(10,)))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer='adam', loss='binary_crossentropy',
                  metrics=['accuracy'])
```

```
    return model

modelo_sem = criar_modelo_sem_protecao()
hist_sem = modelo_sem.fit(
    X_train_norm, y_train,
    epochs=300, batch_size=16,
    validation_split=0.2, verbose=0
)
```

Agora gere o gráfico comparativo lado a lado:

```
def plotar_comparacao(hist_sem, hist_reg):
    """Compara Antes (overfitting) vs Depois (regularizado)."""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # Gráfico 1: SEM proteção
    ax1.plot(hist_sem.history['loss'], 'b-',
             label='Loss Treino')
    ax1.plot(hist_sem.history['val_loss'], 'r-',
             label='Val Loss')
    ax1.set_title('SEM Regularização (Overfitting)')
    ax1.set_xlabel('Épocas')
    ax1.set_ylabel('Loss')
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Gráfico 2: COM Dropout + L2 + Early Stopping
    ax2.plot(hist_reg.history['loss'], 'b-',
             label='Loss Treino')
    ax2.plot(hist_reg.history['val_loss'], 'r-',
             label='Val Loss')
    ax2.set_title(f'COM Regularização (parou na época '
                  f'{len(hist_reg.epoch)})')
    ax2.set_xlabel('Épocas')
    ax2.set_ylabel('Loss')
    ax2.legend()
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('modelo_consertado.png', dpi=150)
    plt.close()
    print("Gráfico salvo: modelo_consertado.png")
```

B. Tabela Comparativa Esperada

Ao rodar o script, você deve observar números semelhantes a estes:

Métrica	Sem Proteção (Lab 05)	Com Regularização (Lab 06)
Épocas executadas	300 (todas)	≈60–120 (Early Stopping)
Acurácia no treino	≈99%	≈82–88%
Acurácia na validação	≈75–80%	≈80–85%
Gap (treino − val)	≈20%	≈2–5%
Loss treino final	≈0.02	≈0.35
Loss validação final	≈0.70	≈0.40
Tempo de treino	Longo (300 épocas)	Curto (Early Stop)

Observe o paradoxo: o modelo regularizado tem acurácia **menor** no treino (não decora!), mas acurácia **maior** na validação (generaliza!). Isso é exatamente o que queremos em produção.

13 Parte 7: Bloco Principal e Execução (5 min)

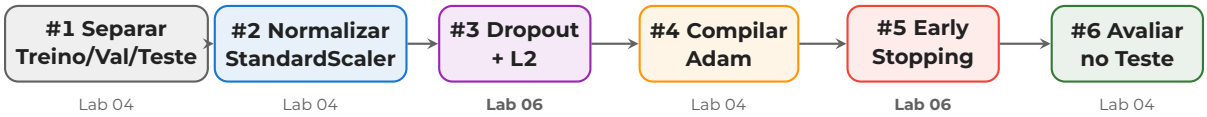
Monte o bloco principal com o relatório completo:

```python
if __name__ == "__main__":
    plotar_comparacao(hist_sem, hist_reg)

    epocas_exec = len(hist_reg.epoch)
    _, acc_reg = modelo_reg.evaluate(
        X_test_norm, y_test, verbose=0)
    _, acc_sem = modelo_sem.evaluate(
        X_test_norm, y_test, verbose=0)

    print(f"\n{'='*50}")
    print(f"  MODELO SEM PROTEÇÃO (Lab 05):")
    print(f"    Épocas: 300 (todas)")
    print(f"    Acc Teste: {acc_sem:.2%}")
    print(f"")
    print(f"  MODELO REGULARIZADO (Lab 06):")
    print(f"    Épocas: {epocas_exec} (Early Stopping)")
    print(f"    Acc Teste: {acc_reg:.2%}")
    print(f"{'='*50}")
```

Execute `python src/modelo_regularizado.py` e verifique que o Early Stopping parou antes de 500 épocas e que as curvas do gráfico caminham juntas.

14 Parte 8: A Receita Profissional (Referência)

Ao completar este lab, você dominou todos os passos da receita profissional de Machine Learning:



Esta receita é válida para **qualquer** problema de classificação com dados tabulares. É a base que você usará no **TP1** do módulo.

15 Parte 9: Garantia de Qualidade — Testes CI/CD (5 min)

Os testes já estão prontos no arquivo `tests/test_regularizado.py` do seu repositório:

```python
import numpy as np
from src.modelo_regularizado import (
    criar_modelo_regularizado, criar_modelo_sem_protecao,
    X_train_norm, y_train, X_test_norm, y_test,
    hist_reg, hist_sem, modelo_reg, vigilante
)

def test_early_stopping_ativou():
    """Early Stopping DEVE ter parado antes de 500."""
    epocas = len(hist_reg.epoch)
    assert epocas < 500, \
        f"Early Stopping falhou: {epocas} épocas"

def test_gap_reduzido():
    """Gap treino-validação menor com regularização."""
    loss_reg = hist_reg.history['loss'][-1]
    val_loss_reg = hist_reg.history['val_loss'][-1]
    gap_reg = abs(val_loss_reg - loss_reg)

    loss_sem = hist_sem.history['loss'][-1]
    val_loss_sem = hist_sem.history['val_loss'][-1]
    gap_sem = abs(val_loss_sem - loss_sem)

    assert gap_reg < gap_sem, \
        f"Gap não reduziu: {gap_reg:.4f} >= {gap_sem:.4f}"

def test_modelo_funcional():
    """Previsões devem estar entre 0 e 1."""
    preds = modelo_reg.predict(X_test_norm, verbose=0)
    assert preds.shape == (len(X_test_norm), 1)
    assert np.all(preds >= 0) and np.all(preds <= 1)

def test_callback_configurado():
    """restore_best_weights DEVE estar ativo."""
    assert vigilante.restore_best_weights == True
```

1. Rode os testes no terminal: `pytest tests/test_regularizado.py -v`.
2. Se a tela exibir **4 passed** em verde, sua implementação está correta.

16 Parte 10: Commit e Push (5 min)

Com todos os testes passando localmente, efetue o envio para o GitLab:

```
git add src/modelo_regularizado.py modelo_consertado.png
git commit -m "Lab 06: Dropout + L2 + Early Stopping vs Overfitting"
git push origin main
```

O GitLab CI interceptará seu push e executará o `pytest` automaticamente. O **Selo Verde** confirma sua pontuação.

△ Importante: Se o Pipeline Falhar

Os erros mais comuns são: (1) esquecer o import `from tensorflow.keras.regularizers import l2`, (2) não passar `callbacks=[vigilante]` no `fit()`, (3) esquecer `restore_best_weights=True` no EarlyStopping. O CI roda os **mesmos testes** que você rodou localmente.

Conclusão: O que Você Construiu Hoje

✔ Takeaways

- Você aplicou **Dropout (30%)** entre camadas densas, impedindo que neurônios se co-adaptem e memorizem ruído — cada neurônio agora é "independente" e útil sozinho.
- Adicionou **Regularização L2** ($\lambda = 0.001$) que penaliza pesos grandes na função de perda: $\mathcal{L}_{total} = \mathcal{L}_{dados} + \lambda \sum w_i^2$.
- Configurou **Early Stopping** com `patience=15` e `restore_best_weights=True`, automatizando a parada ótima e restaurando os pesos do melhor momento.
- Comparou visualmente o gráfico **Antes (Overfitting)** vs **Depois (Regularizado)**, comprovando que as curvas agora caminham juntas e o gap é mínimo.
- Dominou a **receita profissional de 6 passos** para treinar qualquer modelo de classificação tabular robusto.

◇ **Informação**

Gancho para a Próxima Aula: Até agora, todos os seus modelos processaram **tabelas numéricas** (X com linhas e colunas). Mas a IA do mundo real precisa enxergar **imagens**: fotos, radiografias, satélites. Uma imagem 256×256 RGB tem $256 \times 256 \times 3 = 196.608$ pixels — se aplicássemos um `Dense` diretamente, teríamos **milhões de parâmetros** e Overfitting instantâneo. Na próxima aula, entraremos no universo das **Redes Convolucionais (CNNs)**, que exploram a *estrutura espacial* dos pixels para aprender com ordens de grandeza menos parâmetros, e do **Transfer Learning** — onde aproveitamos redes pré-treinadas por Google e Meta para resolver problemas com *poucos dados*.