Teoria de Deep Learning

Vinicius Cantanhede dos Santos

February 2025

1 Introduction

Deep Learning é uma área em constante evolução que tem revolucionado diversos campos da ciência e tecnologia. Neste trabalho, aprofundaremos na parte teórica para um melhor entendimento de tudo que ocorre por trás das redes neurais.

2 Otimização de Buscas

O problema fundamental a ser tratado em redes neurais é a arquitetura (ou, de forma mais diereta, o tamanho da rede neural). A razão ideal entre o número de nós (neurônios) e o número de camadas em uma rede de deep learning não é fixa e depende de vários fatores, como o tipo do probelma, o tamanho da complexidade dos dados, e o objetivo da rede. No entanto, existem diretrizes gerais para ajudar a determinar essa relação:

2.1 Número de camadas

- **Redes Rasas (com poucas camadas):** São mais adequadas para problemas simples, como classificação linear ou problema com poucas variações nos dados.
- **Redes Profundas (com muitas camadas):** São melhores para capturar relações hierárquicas em problemas complexos, como visão computacional e processamento de linguagem natural

2.2 Números de nós por camada

O número de nós em uma camada deve ser suficientemente grande para representar as características dos dados mas não tão grande ao ponto de causar overfitting ou desperdício computacional. É comum começar com mais nós nas camadas iniciais e diminuir nas camadas seguintes. Por exemplo: [128, 64, 32]. Em redes profundas, pode-se usar números constantes ou decrescentes, como [512, 256, 128, 64].

3 Boas Práticas

3.1 Balancear Nós e Camadas

- Para dados mais complexos, prefira redes mais profundas com menos neurônios por camada.
- Para dados mais simples, redes rasas com mais neurônios podem ser suficientes.

3.2 Evite redes excessivamente grndes

- Redes com muitas camadas ou nós podem ser dificéis de treinar devido desvanecimento ou explosão do gradiente.
- Use técnicas como regularização (L1, L2, Dropout) para controlar o overfitting.

3.3 Proporção Empírica

- Comece com 2 a 4 camadas e ajuste o número de nós para ser entre 2 a 10 vezes o número de recursos de entrada.
- Exemplo: Se os dados têm 20 caracteríticas, teste com camadas [64, 32, 16].

3.4 Validação

- Use validação cruzada e ajuste os hiperparametros com base no desempenho do modelo.

3.5 Técnicas de otimização

- Utilize buscas automáticas como GridSearch ou RandomSearch para encontar a melhor combinação de camadas e nós.

4 Random Search

Random Search é uma técninca de otimização para encontrar os melhores hiperparametros de um modelo de machine ou deep learning. Ele seleciona aleatoriamente combinações de hiperparametros dentre do intervalos definidos e avalia o desempenho de cada combinação. É mais eficiente que o Grid Search em muitos casos, especialmente quando alguns hiperparametros têm maior impacto no desempenho do modelo.

4.1 Passos para utilizar o Random Search

4.1.1 Definir os hiperparâmetros e seus intervalos

Liste os hiperparâmetros do modelo que você deseja otimizar e especifique os intervalos ou distribuições de valores possíveis. Por exemplo:

- Número de neurônios = [32, 64, 128, 256]
- Learning Rate = [0.0001, 0.001, 0.01]
- Drop out = [0.1, 0.5]
- Número de camadas = [2, 3, 4]

4.2 Definir o número de Iterações:

Escolha quantas combinações de hiperparametros você deseja testar. Por exemplo, 50 iterações significam que 50 conjuntos de hiperparametros serão testados.

4.3 Treinar e avaliar o modelo

Para cada combinação:

- Configure o modelo de hiperparametros selecionados
- Treine o modelo nos dados de treino
- Avalie o desempenho nos dados de validação usando métricas relevantes, como acurácia, loss ou F1-Score.

4.4 Selecionar os melhores hiperparametros

Após todas as iterações, escolha a combinação de hiperparametros que resultou no melhor desempenho nos dados de validação.

5 Exemplo prático de como utilizar o Random Search

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import random

# Criar um dataset fictício
```

```
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=t
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.lon

# Definir o modelo PyTorch
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_rate):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, 2)  # Saída com 2 classes

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Função para treinar e avaliar o modelo
def train_and_evaluate(params):
    # Desempacotar os parâmetros
    hidden_size = params["hidden_size"]
    learning_rate = params["learning_rate"]
    dropout_rate = params["dropout_rate"]
    batch_size = params["batch_size"]
    num_epochs = params["num_epochs"]

    # Criar o modelo e definir a função de perda e otimizador
    model = SimpleNN(input_size=20, hidden_size=hidden_size, dropout_rate=dropout_rate)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Treinamento
    model.train()
    for epoch in range(num_epochs):
        for i in range(0, len(X_train), batch_size):
            x_batch = X_train[i : i + batch_size]
            y_batch = y_train[i : i + batch_size]

            optimizer.zero_grad()
            outputs = model(x_batch)
            loss = criterion(outputs, y_batch)
```

```
            loss.backward()
            optimizer.step()

    # Avaliação
    model.eval()
    with torch.no_grad():
        outputs = model(X_val)
        _, predictions = torch.max(outputs, 1)
        accuracy = (predictions == y_val).float().mean().item()
    return accuracy

# Espaço de busca de hiperparâmetros
param_grid = {
    "hidden_size": [32, 64, 128],
    "learning_rate": [1e-4, 5e-4, 1e-3],
    "dropout_rate": [0.1, 0.3, 0.5],
    "batch_size": [16, 32, 64],
    "num_epochs": [5, 10, 15],
}

# Random Search
best_score = 0
best_params = None
for _ in range(10):  # Testar 10 combinações aleatórias
    params = {k: random.choice(v) for k, v in param_grid.items()}
    score = train_and_evaluate(params)
    if score > best_score:
        best_score = score
        best_params = params

print("Melhores hiperparâmetros:", best_params)
print("Melhor acurácia:", best_score)
```

6 Otimização Bayesiana

A otimização Bayesiana é uma técninca poderosa para encontrar o máximo ou o mínimo de uma função desconhecida, especialmente em contextos onde a avaliação dessa função é cara ou demorada. Essa abordagem se destaca em várias aplicações, como otimização de hiperparametros em modelos de aprendizado de máquina, exeperimentos científicos e processos industriais. Utiliza métodos probabilísticos para modelar o função de custo e encontrar melhores combinações de hiperparametros com menos iterações.

6.1 Princípios Fundamentais

A otimização bayesiana baseia-se na teoria bayesiana, que envolve a utilização de crenças a medida que novas evidencias se tornam disponíveis. A técnica utiliza um modelo probabilístico, frequentemento um processo gaussiano, para representar a função objetivo. Esse modelo é continuamente atualizados com os resultados das avaliações anteriores, permitindo que a otimização se concentre nas áreas do espaço de busca que têm maior probabilidade de conter o ótimo global.

6.2 Contexto da otimização de hiperparametros

A otimização bayesiana é uma técnica amplamente utilizada na otimização de hiperparametros em redes neurais, devido a sua eficiência em explorar espaços de buscas e encontrar consigurações que maximizam o desempenho do modelo. A escolha adequada dos hiperparametros pode ter um impacto significativo no desempenho da rede, mas a busca por combinações ideais pode ser complexa e demorada, especialmente considerando que cada avaliação pode exigir treinamento completo do modelo.

6.3 Processo de Otimização Bayesiana para Hiperparametros

- **Definição do espaço de busca:** Inicialmente, define-se um espaço de busca para os hiperparametros, especificando os intervalos ou categorias possíveis de cada um.
- **Modelo Probabilístico:** Um modelo probabilístico é criado para estimar a função objetivo, que neste caso é a métrica de desempenho de uma rede neural (como precisão ou perda) em relação aos hiperparâmetros.
- **Avaliação inicial:** Realizam-se algumas avaliações iniciais aleatórias dos hiperparametros. Os resultados dessa avaliações são utilizados no modelo probabilísitico.
- **Função de aquisição:** Uma função de aquisição é utilizada para determinar quais hiperparametros devem ser testados a seguir. Essa função equilibra a exploração baseado na incerteza do modelo e no potencial de melhorar os resultados.
- **Atualização do modelo:** Após cada nova avaliação dos hiperparametros, o modelo probabilístico é atualizado com os novos dados, melhorando suas previsões sobre o desempenho da rede neural.
- **Iteração:** O processo se repete até que um critério de parada seja atendido, como um número máximo de avaliações ou uma melhoria mínima no desempenho.

6.4 Vantagens da Otimização Bayesiana na Otimização de Hiperparâmetros

- **Eficiência:** Ao contrário de abordagens como busca em grade ou aleatória, a otimização bayesiana requer menos avaliações para encontrar configurações eficazes, economizando tempo e recursos computacionais.
- **Manejo da incerteza:** A técnica permite quantificar a incerteza nas previsões sobre o desempenho dos hiperparâmetros, levando a decisões mais informadas sobre onde explorar.
- **Melhor exploração do espaço:** A função de aquisição ajuda a direcionar as buscas para regiões promissoras do espaço de hiperparâmetros, evitando áreas que já foram testadas sem sucesso.

7 Exemplo prático de como utilizar a Otimização Bayesiana

```
import optuna
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Criar dataset
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

X_train, y_train = torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=t
X_val, y_val = torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.lon

# Modelo PyTorch
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, dropout_rate):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, 2)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
```

```
        x = self.fc2(x)
        return x

# Função objetivo para Optuna
def objective(trial):
    # Sugerir hiperparâmetros
    hidden_size = trial.suggest_categorical("hidden_size", [32, 64, 128])
    learning_rate = trial.suggest_loguniform("learning_rate", 1e-4, 1e-2)
    dropout_rate = trial.suggest_uniform("dropout_rate", 0.1, 0.5)
    batch_size = trial.suggest_categorical("batch_size", [16, 32, 64])
    num_epochs = trial.suggest_int("num_epochs", 5, 15)

    # Criar o modelo
    model = SimpleNN(input_size=20, hidden_size=hidden_size, dropout_rate=dropout_rate)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Treinamento
    model.train()
    for epoch in range(num_epochs):
        for i in range(0, len(X_train), batch_size):
            x_batch = X_train[i : i + batch_size]
            y_batch = y_train[i : i + batch_size]

            optimizer.zero_grad()
            outputs = model(x_batch)
            loss = criterion(outputs, y_batch)
            loss.backward()
            optimizer.step()

    # Avaliação
    model.eval()
    with torch.no_grad():
        outputs = model(X_val)
        _, predictions = torch.max(outputs, 1)
        accuracy = (predictions == y_val).float().mean().item()

    return accuracy

# Otimizar com Optuna
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=20)  # Realizar 20 experimentos

# Resultados
print("Melhores hiperparâmetros:", study.best_params)
print("Melhor acurácia:", study.best_value)
```

8 Introdução de teoria de Deep Learning

Notações:

- L é o número de camadas ocultas.
- l é uma camada oculta específica.
- x é o input, um vetor n de dimensão 0.
- σ é uma função de ativação.
- $w^{(l)}$ são os pesos das matrizes .
- N_l é o tamanho da l-ésima camada oculta.

Inferindo que uma rede Z com camadas totalmente conectadas pode ser representada como:

(1)

onde cada w^{l_i} incorpora x em um espaço n unidimensional e aplica σ a cada elemento do vetor resultante $x_{w^l\sigma}$, transmitindo-o a próxima $w^{l_{i+1}}$. E, de forma extensiva, a matriz de pesos pode ser representada como:

(2)

9 Q1. Por que a otimização funciona mesmo quando a loss não é convexa?

9.1 Super-parametrização:

Número de parâmetros de um modelo (pesos, bias) >>> Número de pontos de dados (uma única observação) >>> 1.

9.2 Abordagem 1:

Encontrar a melhor combinação de parâmetros por meio da busca ao longo do espaço de parâmetros possíveis pode ser expressado por:

$$ls = \frac{\sin\left(3\sqrt{x^2y^2}\right)}{\left(\sqrt{x^2y^2} + 1.5\right)\,\sin x\,\cos x}$$

Considerando, de forma simplificada, que apenas dois parâmetros $p1$ e $p2$ atual sobre o modelo, o espaço de parâmetro teria o mapa de calor a seguir:

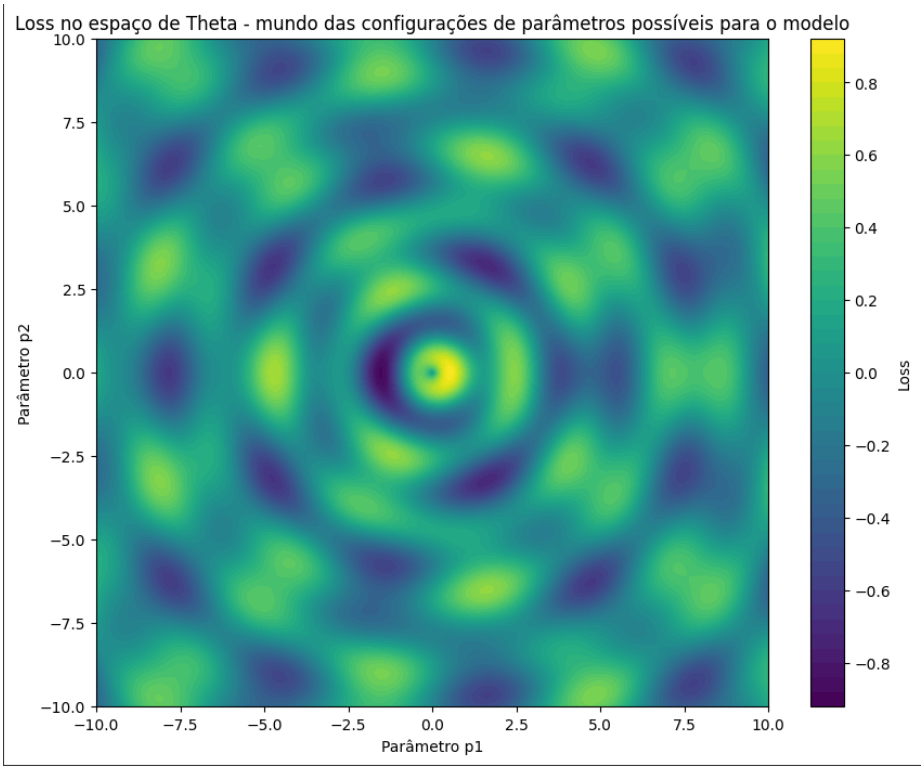

Figure 1: Loss no espaço Theta

Nesse contexto, caso pegássemos os pares [-8, -8], [-4, -4], [0, 0], [4, 4], [8, 8] para $p1$ e $p2$, respectivamente, teríamos as seguindes posições de cada par no espaço de busca:

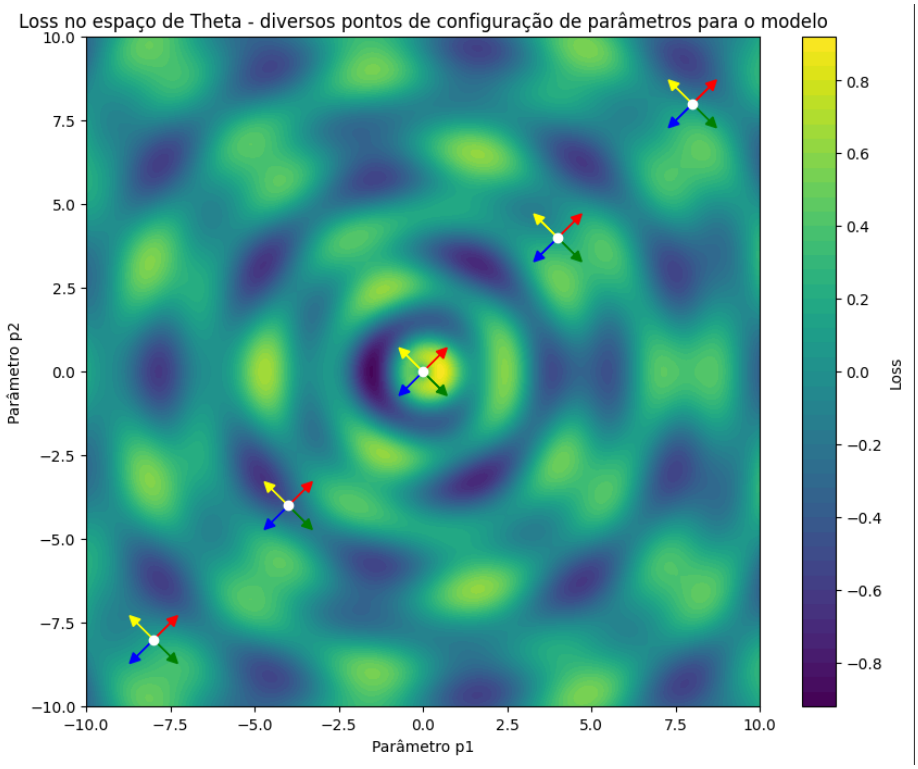

Figure 2: Loss no espaço Theta - Diversos pontos de configuração de parâmetros para o modelo

10 Transpondo o problema de mapeamento de parâmetros para mapeamento de predições por meio dos dados de treinamento

Considerando que o espaço de parâmetros é muito maior que o número de pontos de dados coletados e/ou observados, reduzir o espaço de busca pode levar a um melhor resultado, economizando não só tempo, mas também encontrando melhores configurações de parâmetros. Para se empreender a esse tipo de busca por soluções, **é necessário deixar de pensar em quais são os conjutos de parâmetros a serem configurados e passar a observar os valores obtidos ao longo das etapas de treinamento de rede**. Desta forma, consideremos:

- Um mapa de parâmetros ω;
- ω mapeia o vetor de parâmetros θ para um conjunto de valores $(x_i; \theta)$ que o modelo Z assume ao processar os pontos de dados de treinamento;

- Desde que $i = 1$ até $i = n$ de pontos de dados.

$$\omega \to Z^{(L+1)}(x_i; \theta);$$

Isso possbilita analisarmos cada configuração de parâmetros por meio de resultados obtidos a partir dos dados de treinamento. Considerando que a **função de LOSS** das saídas das redes são convexas, podemos deduzir que o melhor caminho seria seguir o caminho do gradiente dessa função.

11 Premissa 1:

Quando o número de parâmetros é muito maior que o número de pontos de dados, a matriz Jacobiana do mapa de parâmetros de uma configuração θ específica é de posto completo.

11.1 O que é uma matriz jacobiana?

A matriz Jacobiana é uma ferramenta fundamental em cálculo diferencial e análise de funções de várias variáveis. Se trata de uma matriz composta pelas derivadas parciais de uma função vetorial multivariada, representando a melhor aproximação linear de uma função em torno de um ponto. Considere uma função:

$$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m, \quad \mathbf{f}(x_1, x_2, \ldots, x_n) = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ f_2(x_1, \ldots, x_n) \\ \vdots \\ f_m(x_1, \ldots, x_n) \end{pmatrix}.$$

A matriz jacobiana de $\mathbf{f}$, denotada por $J_{\mathbf{f}}(x)$ ou simplesmente $\mathbf{J}(x)$, é dada por

$$J_{\mathbf{f}}(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \frac{\partial f_1}{\partial x_2}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \frac{\partial f_2}{\partial x_1}(x) & \frac{\partial f_2}{\partial x_2}(x) & \cdots & \frac{\partial f_2}{\partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \frac{\partial f_m}{\partial x_2}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix}.$$

Cada elemento $\frac{\partial f_i}{\partial x_j}$ desta matriz representa a taxa de variação da i-ésima componente de $\mathbf{f}$ em relação à j-ésima variável. Essa estrutura é especialmente útil para:

- **Linearização:** Aproximar a função perto de um ponto por uma função linear.
- **Teorema da Função Inversa:** Analisar as condições para a inversibilidade local de uma função.

- **Mudança de Variáveis:** No cálculo de integrais múltiplas, o determinante da matriz jacobiana é utilizado para ajustar o fator de escala na transformação.

A matriz Jacobiana generaliza a derivada para função multivariadas, descrevendo como pequenas variações nas entradas afetam as saídas. Seu determinante quantifica mudanças de volume e seu posto revela propiedades locais da função.

11.2 Voltando a premissa 1...

Razões para essa premissa: Dado que o número de parâmetros é muito maior que o número de dados, alguns parâmetros trabalham para o ajustes de pontos de dados, outros parâmetros trabalham para o ajuste de outros pontos de dados. Essa possibilidade diminui a concorrência entre pontos de dados nos ajustes dos parâmetros.
Assumindo que a matriz jacobiana de uma configuração tem posto completo: Qualquer direção que se deseje ir em uma configuração específica de parâmetros, há uma direção correspondente no espaço de todos os parâmetros possíveis. Em consequência do posto completo da jacobiana, pode-se concluir que há somente um mínimo para a função.
Definição de posto de uma matriz: O posto (ou rank) de uma matriz A é o maior número de colunas (ou linhas) linearmente independentes na matriz. Ou seja:

- Posto das colunas: É o número máximo de colunas linearmente independentes.
- Posto das linhas: É o número máximo de linhas linearmente independentes.

Exemplo:
Considere a matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}.$$

Podemos analisar os vetores linha (ou colunas) de A. Neste exemplo, vamos considerar os vetores linha:

$$v_1 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \quad v_2 = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}, \quad v_3 = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix}.$$

A ideia é verificar se algum desses vetores podem ser escritos como combinação linear dos outros. Se for possível expressar, por exemplo, v_3 em termos de v_1 e v_2, isso indica que os três vetores não são linearmente independentes.
Procuramos escalares a e b tais que

$$v_3 = a\, v_1 + b\, v_2.$$

Ou seja,

$$\begin{pmatrix}7\\8\\9\end{pmatrix} = a\begin{pmatrix}1\\2\\3\end{pmatrix} + b\begin{pmatrix}4\\5\\6\end{pmatrix}.$$

Isso gera o sistema de equações:

$$\begin{cases}7 = a\cdot 1 + b\cdot 4,\\ 8 = a\cdot 2 + b\cdot 5,\\ 9 = a\cdot 3 + b\cdot 6.\end{cases}$$

Da primeira equação temos:

$$a + 4b = 7 \quad\Longrightarrow\quad a = 7 - 4b.$$

Substituindo essa expressão na segunda equação:

$$2(7-4b) + 5b = 8 \quad\Longrightarrow\quad 14 - 8b + 5b = 8,$$

$$14 - 3b = 8 \quad\Longrightarrow\quad -3b = -6 \quad\Longrightarrow\quad b = 2.$$

Agora, substituindo $b = 2$ na expressão para a:

$$a = 7 - 4(2) = 7 - 8 = -1.$$

Para confirmar, verificamos a terceira equação:

$$-1\cdot 3 + 2\cdot 6 = -3 + 12 = 9,$$

o que é consistente com v_3.
Portanto, concluímos que

$$v_3 = -v_1 + 2v_2.$$

Isso mostra que v_3 é uma combinação linear dos vetores v_1 e v_2. Assim, os três vetores não são linearmente independentes, e o espaço gerado por eles é, na verdade, bidimensional.
Em outras palavras, a matriz A possui apenas 2 vetores linearmente independentes, ou seja, o **posto** (ou *rank*) de A é 2.

12 Como se realiza o feature learning? Como endereçar o problema da dimensionalidade?

INTRO: Regime NTK de treinamento (ou lazy regime, kernel regime, linear regime) Conisdere como constantes:

- quantidade de dados de treinamento;
- a dimensão de entrada da rede como N_0;

- a dimensão dos dados de saída N_{L+1};
- a profundidade da rede neural L, sendo $L \geq 1$

Inicializa-se o treinamento da rede, com uma configuração particular (mesma forma utilizada pelo pytorch):

$$W_{i,j}^{l} \sim \mathcal{N}(0, \frac{C_W}{N_{l-1}})$$

- Pesos iniciados com distribuições gaussianas independentes;
- A média dos pesos é igual a zero
- A variação é escalonada de forma constante C_w baseada da largura de camada anterior N_{l-1}.

Baseado nessas configurações, conforme a largura das camadas escondidas aumenta:

- No início, a rede neural convergirá em processos gaussianos com neurônios independentes, ou seja, não haverá correlação entre neurônios nem uma propensão para calcular a correção entre eles.
- Durante a otimização, o fluxo de gradiente (descida) $\mathcal{L}$, é possível substituir a rede neural não-linear por sua linearização:

$$\frac{du}{dt}\theta(t) = -\nabla_\theta \mathcal{L}(\theta(t)), então$$

$$Z^{L+1}(\theta) \to Z^{lin}(\theta) = Z^{(L+1)}(\theta(0)) + \langle \nabla_\theta Z(\theta(0)), \theta - \theta(0) \rangle$$

- $Z^{(L+1)}(\theta(0))$ é configuração inicial da rede;
- $\langle \nabla_\theta Z(\theta(0)), \theta - \theta(0) \rangle$ é o termo de primeira ordem da expansão de Taylor, a partir da configuração inicial.

Ou seja:

- Ao aumentar a largura das camadas escondidas de uma rede não-linear, seu comportamento tenderá um modelo linear.
- A matriz jacobiana de uma rede linear é constante.
- Requer-se, nesse caso, que tão somente a matriz jacobiana na inicialização seja de posto completo.

Esse regime afasta a possibilidade de feature learning, dado que seria necessário tão somente que a jacobiana inicial seja de posto completo.

13 Premissa 2: Há formas de sobrepujar o regime NTK

1. Alterar técnicas de aprendizado:

   - Mudar a fórmula de cálculo da LOSS.
   - Manter o step size de busca maior.
   - Inserir técnicas de regularização (L1, L2, Dropout, Elastic Net)

2. Considerar que os pontos dos dados aumentam conforme a largura da rede aumenta.
Meta-teorema: Proporcionalidade entre largura e profundidade de uma rede.
Quando as camadas N_l são largas, mas finitas (a rede é profunda mas não tende ao infinito), a distância entre Z^L e Z^{lin} é a razão $\frac{L}{N}$, como consequência de que a correlação entre os neurônios também se traduz por $\frac{L}{N}$ e a variância dos gradientes $|\nabla_\theta Z(\theta)||^2$ tambpem cresce em razão $\frac{N}{L}$.
Conclui-se, desta forma, que, sendo $\frac{N}{L} = \beta$:

   - A melhor forma de se ajustar a rede é tender tanto sua largura N quanto sua profundidade L ao infinito, em uma proporção fixa de crescimento (e não fixar uma e teder ao infinito outra);
   - A medida em que β aumeta, desenvolve-se feature learning, ou seja, se está mais longe de um modelo linear Z^{lin}, mas mais tendente a explosão e desaparecimento de gradientes.
   - A medida em que β diminui, desenvolve-se a otimização, ou seja, se está mais estável para realizar a descida do gradiente, mas aproxima-se da rede linear Z^{lin}, tendendo a overfitting.