Natural Language Processing (NLP) and Large Language Models (LLMs)
Lecture 3-1: Machine learning II

Chendi Wang (王晨笛)
chendi.wang@xmu.edu.cn

WISE @ XMU

2025 年 3 月 13 日

Adapted in part from 《动手学深度学习》，《AI and machine learning》，and CS 224n.

Many NLP Tasks are Classification Problems

- Classify sentences into positive, negative, or neutral sentiment (sentiment analysis).
- Given a center word and its surrounding context, classify the center word into categories such as Person, Location, Organization, or Other.

Named-Entity Recognition

- Named-Entity Recognition (NER): find and classify names in text.
- Kobe Bryant was a legendary basketball player for the Los Angeles Lakers from 1996 to 2016, after completing his high school education in Philadelphia.
    - Kobe Bryant: Person
    - Los Angeles Lakers: Organization
    - 1996 to 2016: Time
    - Philadelphia: Location

Classification notation

- Input: $\mathbf{x} \in \mathbb{R}^d$ is a word vector, or a sentence, or a document etc.
- Label: $y \in \{1, 2, \cdots, K\}$
    - sentiment:(1= positive, 2= negative)
    - named entities, 1 = Person; 2= Time, ...
    - decision: 1=buy, 2= sell
- Onehot label: $\mathbf{y} = [y^{(1)}, \cdots, y^{(K)}] = [0, \cdots, 0, 1, 0, \cdots, 0]$ ($y^{(k)} = 1$ if $y = k$).

Window Classification

- Goal: Classify each word based on its context window.
- Consider a sequence of word embeddings within a sentence:

$$\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}\}, \quad \text{each } \mathbf{x}^{(t)} \in \mathbb{R}^d.$$

- For each word $\mathbf{x}^{(t)}$, define a context window of size $2K+1$:

$$\mathbf{x}^{(t-K)}, \ldots, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t)}, \mathbf{x}^{(t+1)}, \ldots, \mathbf{x}^{(t+K)}.$$

- Construct input vector by concatenating embeddings in the window:

$$\mathbf{x}_t = [\mathbf{x}^{(t-K)}, \ldots, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t)}, \mathbf{x}^{(t+1)}, \ldots, \mathbf{x}^{(t+K)}] \in \mathbb{R}^{(2K+1)d}.$$

- Label: y_t is the class label for the **center word** $\mathbf{x}^{(t)}$.
- Training data: $\{(\mathbf{x}_t, y_t)\}_{t=1}^{T}$

Training

- Training data: $\{\mathbf{z}_i = (\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$.
- For a classifier (such as a neural network or a linear classifier) $\mathbf{f}_\theta = (f_\theta^{(1)}, \cdots, f_\theta^{(K)})$, find θ that minimizes:

$$\frac{1}{n}\sum_{i=1}^n \ell_\theta^{\mathrm{CE}}(\mathbf{z}_i) = \frac{1}{n}\sum_{i=1}^n \sum_{j=1}^K y_i^{(j)} \log f_\theta^{(j)}(\mathbf{x}_i).$$

Neural networks update the embeddings

- Recall

$$\mathbf{f}_\theta(\mathbf{x}) = \mathrm{SoftMax}\left(\mathbf{W}^{[L]}\sigma(\mathbf{W}^{[L-1]}\sigma(\cdots\sigma(\mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^{[1]}))+\mathbf{b}^{[L-1]})\right).$$

- Rewrite

$$\mathbf{f}_\theta(\mathbf{x}) = \mathrm{SoftMax}\left(\mathbf{W}^{[L]}\mathbf{h}(\mathbf{x})\right)$$

  with $\mathbf{h}(\mathbf{x}) = \sigma(\mathbf{W}^{[L-1]}\sigma(\cdots\sigma(\mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^{[1]}))+\mathbf{b}^{[L-1]})$.

- $\mathbf{h}(\mathbf{x}) \in \mathbb{R}^{d^{[L]}}$ is a new word vector (a.k.a., embedding, representation, or feature for image data).
- Note that $\mathbf{h}$ is updated when θ is updated.

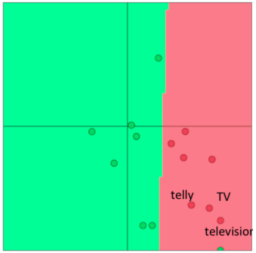
Training

- Training data: $\{\mathbf{z}_i = (\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$.
- For a classifier (such as a neural network or a linear classifier) $\mathbf{f}_\theta = (f_\theta^{(1)}, \cdots, f_\theta^{(K)})$, find θ that minimizes:

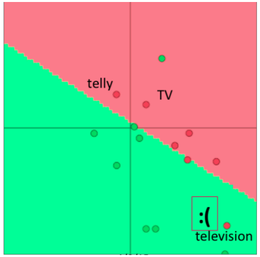
$$\frac{1}{n}\sum_{i=1}^n \ell_\theta^{\mathrm{CE}}(\mathbf{z}_i) = \frac{1}{n}\sum_{i=1}^n \sum_{j=1}^K y_i^{(j)} \log f_\theta^{(j)}(\mathbf{x}_i).$$

Retraining word embeddings

- The pretrained word vectors (such as using GloVe) could be further retrained in downstream tasks (such as classification) to perform better.
- However, retraining is risky (see the figure).
- When retraining, we need to ensure that the training set is large enough to cover most words from the vocabulary.



Linear classifiers before retraining



Linears classifier after retraining

An example when linear classifiers fail to classify the word “television” after retraining

Figure is from CS 224n

Recall: Ridge regression

- Linear ridge regression:

$$\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^T x_i - b\right) + \lambda \sum_{j=1}^{d} w_j^2$$

- $\lambda \sum_{j=1}^{d} w_j^2$ is a regularization term that is used to prevent overfitting.
- λ here is a hyperparameter (adjusting bias-variance trade-off).

Overfitting and underfitting

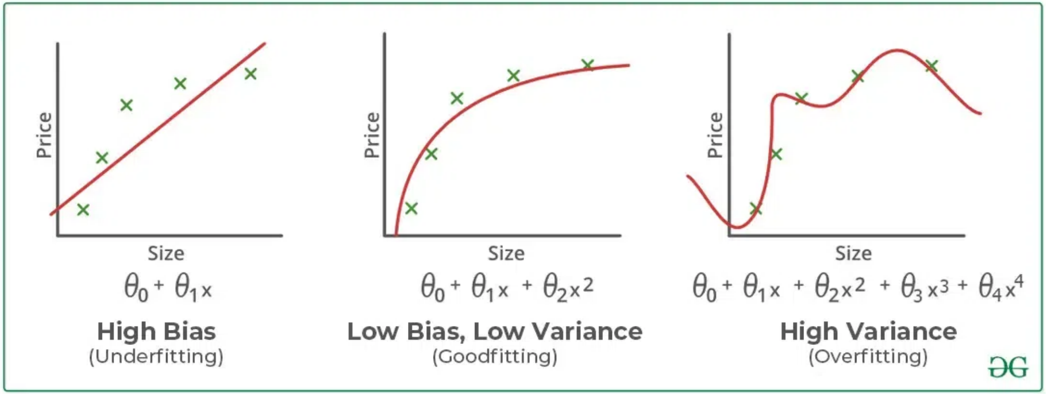


Figure is from https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

Bias-variance trade-off

- For a data point $(\mathbf{x}, y)$ such that $y = f_*(\mathbf{x}) + \epsilon$ with $\mathbb{E}[\epsilon|\mathbf{x}] = 0$, consider an estimate $\widehat{y} = f_{\widehat{\theta}}(\mathbf{x})$ ($\widehat{y} = \widehat{\mathbf{w}}^T\mathbf{x} + \widehat{b}$ for linear models).
- $\widehat{\theta}$ is random since it is trained on training data.
- $\mathbb{E}_{\widehat{\theta},\epsilon}[(y - f_{\widehat{\theta}}(\mathbf{x}))^2] = \mathrm{Bias}^2 + \mathrm{Variance} + \sigma^2$.
- $\sigma^2 = \mathbb{E}\left[\epsilon^2|x\right]$ is called the Bayes error (lowest error rate).
- Bias: $\mathbb{E}_{\widehat{\theta}}[\widehat{y}] - f_*(\mathbf{x})$
- Variance: $\mathrm{Var}\left[\widehat{y}\right] = \mathbb{E}_{\widehat{\theta}}\left[\widehat{y} - \mathbb{E}_{\widehat{\theta}}[\widehat{y}]\right]^2$.
- Bias increases, variance decreases (and vice versa); controlled by λ.
- One can verify that OLS is un-biased, check the variance (and calculate the bias and variance for linear ridge regression).

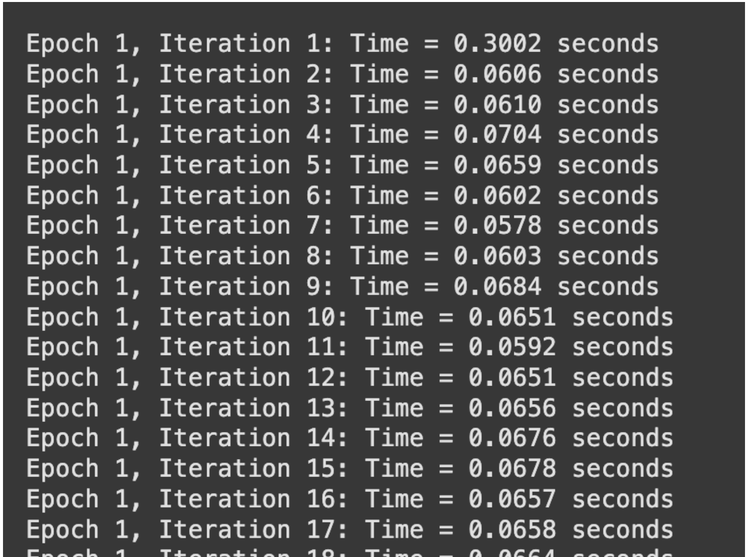
Regularization in deep learning

- In deep learning, we usually don't use explicit regularization.
- Implicit regularization can be achieved by tuning hyperparameters.
- Trainable parameters (estimated by training data): θ
- Hyperparameters can not be estimated using training data.

An example of training a convolutional neural network (CNN)

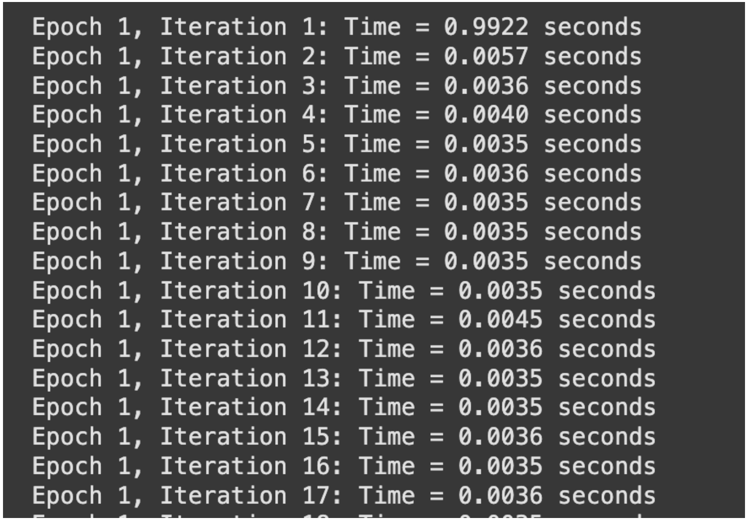
- train-CNN.ipynb
- device:CPU v.s. GPU
- An epoch is a complete pass through the entire training dataset during the training of a machine learning model.

Running time for CPU



```
Epoch 1, Iteration 1: Time = 0.3002 seconds
Epoch 1, Iteration 2: Time = 0.0606 seconds
Epoch 1, Iteration 3: Time = 0.0610 seconds
Epoch 1, Iteration 4: Time = 0.0704 seconds
Epoch 1, Iteration 5: Time = 0.0659 seconds
Epoch 1, Iteration 6: Time = 0.0602 seconds
Epoch 1, Iteration 7: Time = 0.0578 seconds
Epoch 1, Iteration 8: Time = 0.0603 seconds
Epoch 1, Iteration 9: Time = 0.0684 seconds
Epoch 1, Iteration 10: Time = 0.0651 seconds
Epoch 1, Iteration 11: Time = 0.0592 seconds
Epoch 1, Iteration 12: Time = 0.0651 seconds
Epoch 1, Iteration 13: Time = 0.0656 seconds
Epoch 1, Iteration 14: Time = 0.0676 seconds
Epoch 1, Iteration 15: Time = 0.0678 seconds
Epoch 1, Iteration 16: Time = 0.0657 seconds
Epoch 1, Iteration 17: Time = 0.0658 seconds
Epoch 1, Iteration 18: Time = 0.0664 seconds
```

Running time for GPU



```
Epoch 1, Iteration 1: Time = 0.9922 seconds
Epoch 1, Iteration 2: Time = 0.0057 seconds
Epoch 1, Iteration 3: Time = 0.0036 seconds
Epoch 1, Iteration 4: Time = 0.0040 seconds
Epoch 1, Iteration 5: Time = 0.0035 seconds
Epoch 1, Iteration 6: Time = 0.0036 seconds
Epoch 1, Iteration 7: Time = 0.0035 seconds
Epoch 1, Iteration 8: Time = 0.0035 seconds
Epoch 1, Iteration 9: Time = 0.0035 seconds
Epoch 1, Iteration 10: Time = 0.0035 seconds
Epoch 1, Iteration 11: Time = 0.0045 seconds
Epoch 1, Iteration 12: Time = 0.0036 seconds
Epoch 1, Iteration 13: Time = 0.0035 seconds
Epoch 1, Iteration 14: Time = 0.0035 seconds
Epoch 1, Iteration 15: Time = 0.0036 seconds
Epoch 1, Iteration 16: Time = 0.0035 seconds
Epoch 1, Iteration 17: Time = 0.0036 seconds
```

Some hyperparameters

- Learning rate η
- Mini-batch size b (related to learning rate)
- Depth L
- Number of neurons per layer $d^{[l]}, l = 1, \cdots, L$.
- Algorithms: SGD, Adam, Nesterov accelaration etc.
- Number of iterations (early stopping)
- Dropout (set some parameters to be 0)
- Initialization
- Temperature (for generative models) etc.

Initialization

- One simple way is to initialize parameters as $\theta_0 = 0$.
- In train-CNN.ipynb, we use the default method: random initialization, also known as Kaiming (He) initialization.
- **Random Initialization**: For each layer's weight matrix

$$\mathbf{W}^{[l]} = [\mathbf{w}_1^{[l]}, \cdots, \mathbf{w}_{d^{[l+1]}}^{[l]}] \in \mathbb{R}^{d^{[l]} \times d^{[l+1]}},$$

  initialize each column vector independently as

$$\mathbf{w}_i^{[l]} \sim \mathcal{N}(0, \sigma^2 I), \quad \text{with} \quad \sigma^2 = \frac{2}{d^{[l]}}.$$

- Bias terms $\mathbf{b}^{[l]}$ are typically initialized as $\mathbf{0}$.

Early Stopping in Deep Learning

- **Idea:** Stop training when performance on a validation set no longer improves, to prevent overfitting.
- **Procedure:** Track validation loss during training; halt training when validation loss starts increasing or remains unchanged. (Example early-stopping.ipynb)
- **Benefit:** Provides an automatic and practical way to select the number of epochs, improving generalization.
- One can show that early stopping is mathematically similar to ridge regression for linear models.

Dropout

- Consider the hidden layer output $\mathbf{h} \in \mathbb{R}^{d^{[l]}}$ (e.g., $\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ for $l = 1$).
- Define the random masking vector $\mathbf{r} = [r_1, \ldots, r_{d^{[l]}}] \in \{0, 1\}^{d^{[l]}}$, where each r_i independently satisfies:

$$\mathrm{Prob}[r_i = 0] = p_{\mathrm{drop}}.$$

- Dropout applies as:

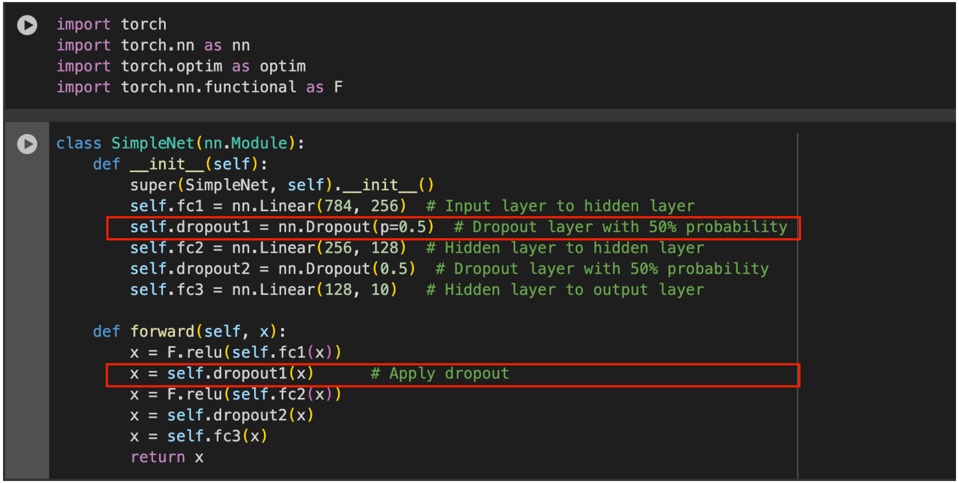
$$\mathbf{h}_{\mathrm{drop}} = \gamma(\mathbf{r} \odot \mathbf{h}),$$

  where $\odot$ denotes the Hadamard (element-wise) product.
- The scaling factor γ ensures unbiasedness:

$$\gamma \mathbb{E}[r_i h_i] = h_i \quad \text{for all } i.$$

- Dropout helps prevent overfitting by randomly setting a fraction of activations to zero, reducing co-adaptation among neurons.

Dropout in pytorch



```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 256)  # Input layer to hidden layer
        self.dropout1 = nn.Dropout(p=0.5)  # Dropout layer with 50% probability
        self.fc2 = nn.Linear(256, 128)  # Hidden layer to hidden layer
        self.dropout2 = nn.Dropout(0.5)  # Dropout layer with 50% probability
        self.fc3 = nn.Linear(128, 10)   # Hidden layer to output layer

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)       # Apply dropout
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

Learning rate

- The magnitude of the learning rate matters!!!!!
- Large learning rate: may not converge
- Small learning rate: converges slowly
- No fixed rule, depending on your task and model (Experimentation and observation are key)
- A common combination: warmup + decay + adaptive methods.

Why is Warm-up Important?

- In the early stages of training, gradients are typically large.
- If the learning rate is high initially, parameters may change dramatically, causing training instability.
- Gradually increasing the learning rate (warm-up) helps stabilize training and enables better convergence.
- It also assists in escaping poor local minima.

What is Warm-up?

- We start from a small learning rate η_{init} and increase it to a relatively large η_{base} in the first T_{warm} steps (or epochs).
- Mathematically, for the t-th step we have

$$\eta_t = \eta_{\text{init}} + (\eta_{\text{base}} - \eta_{\text{init}}) \cdot \frac{t}{T_{\text{warm}}}, \quad 0 \leq t \leq T_{\text{warm}}.$$

- See warmup.ipynb

Learning rate decay (after warm-up)

```
trainer = torch.optim.SGD(net.parameters(), lr=0.1)
scheduler = lr_scheduler.StepLR(trainer, step_size=5, gamma=0.1)
```

- lr $= \eta_{\text{base}}$
- Polynomial decay: the learning rate of Epoch t is $\mathrm{lr} \times \mathrm{gamma}^{\lfloor t/\mathrm{step\_size} \rfloor}$
- Other decay rules: exponential decay, Cosine Annealing etc.

Optimized Hyperparameter Tuning: Grid Search

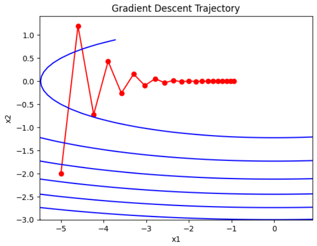
- Grid Search: Evaluates all combinations from a specified set of parameters.
- Advantages: simple to implement and interpret (cause it is optimized over the defined grid); easy parallelization.
- Disadvantages: Computationally expensive (time-consuming); Efficiency decreases exponentially with the number of parameters.
- Common libraries: `scikit-learn`, `Optuna`.

GD may not converge

- Minimize

$$g(\theta_1, \theta_2) = 0.1\theta_1^2 + 2\theta_2^2.$$

- See gd-counter-example.ipynb
- The gradient in the θ_2 direction is much larger and is changing more rapidly than the horizontal direction (θ_1).



Example is from `https://d2l.ai/chapter_optimization/momentum.html`

Recap: SGD

- Loss function:

$$\frac{1}{n}\sum_{i=1}^{n}\ell_\theta(\mathbf{z}_i).$$

- In the t-th iteration, sample $\mathcal{B}_t \subset \{1, 2, \cdots, n\}$ with mini-batch size $|\mathcal{B}_t| = b$
- SGD update $\theta_{t+1} = \theta_t - \frac{\eta_t}{b}\sum_{i\in\mathcal{B}_t}\nabla_\theta \ell_\theta(\mathbf{z}_i)$.
- Rewrite $\mathbf{g}_t = \frac{1}{b}\sum_{i\in\mathcal{B}_t}\nabla_\theta \ell_\theta(\mathbf{z}_i)$ and

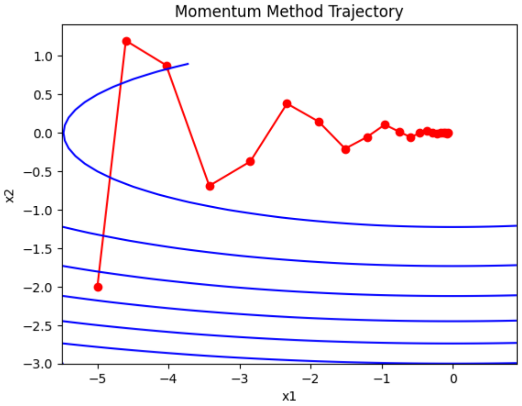
$$\theta_{t+1} = \theta_t - \eta_t \mathbf{g}_t.$$

The momentum method

- Let
$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \mathbf{g}_t.$$
- Updates
$$\theta_{t+1} = \theta_t - \eta_t \mathbf{v}_{t+1}$$
- Momentum replaces gradients with a leaky average over past gradients, that is
$$\mathbf{v}_{t+1} = \sum_{s=0}^{t-1} \beta^s \mathbf{g}_s.$$

Example

- Recall the example in gd-counter-example.ipynb.
- Using momentum method, we have



Adagrad (Duchi et al., 2011)

- Adagrad adapts the learning rate by accumulating past squared gradients using a state vector $\mathbf{s}_t$.
- Denote the gradient at step t by $\mathbf{g}_t = \frac{1}{b}\sum_{j\in\mathcal{B}_t} \nabla_\theta \ell_\theta(\mathbf{z}_j)$.
- **Iteration formula**:

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \quad \theta_t = \theta_{t-1} - \frac{\eta\, \mathbf{g}_t}{\sqrt{\mathbf{s}_t + \delta \cdot \mathbf{1}}}$$

  (All operations are element-wise.)
- Here, δ is a small constant for numerical stability (typically 10^{-6} or 10^{-7}).
- Example gd-counter-example.ipynb (adjusting learning rate)

RMSProp (Tieleman and Hinton, 2012)

- In the original Adagrad algorithm, the accumulator vector $\mathbf{s}_t$ grows without bound, causing the learning rate to decay excessively.
- A common modification is to introduce a decay factor $0 < \gamma < 1$:

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2, \quad \theta_t = \theta_{t-1} - \frac{\eta\, \mathbf{g}_t}{\sqrt{\mathbf{s}_t + \delta \cdot \mathbf{1}}}.$$

(Element-wise operations)

- Example implementation: gd-counter-example.ipynb

Adam (Kingma and Ba; 2015)

- Adam combines the ideas of momentum and RMSProp:

$$\text{Momentum:}\quad \mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\mathbf{g}_t$$

$$\text{RMSProp:}\quad \mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$$

- Bias correction step (to offset initialization at zero):

$$\widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta^t}, \quad \widehat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\gamma^t}$$

(since $1 + \beta + \beta^2 + \cdots + \beta^{t-1} = \frac{1-\beta^t}{1-\beta}$)

- Parameter update rule:

$$\theta_t = \theta_{t-1} - \frac{\eta_t \widehat{\mathbf{v}}_t}{\sqrt{\widehat{\mathbf{s}}_t} + \delta \cdot \mathbf{1}}$$

(all operations element-wise)

- Example implementation: gd-counter-example.ipynb

Adam (Kingma and Ba; 2015)

Train a CNN using Adam: Adam-CNN.ipynb

Weight decay (l_2-penalty)

- l_2-penalty: $\frac{1}{n}\sum_{i=1}^{n}\ell_\theta(\mathbf{z}_i) + \lambda\|\theta\|_2^2$
- SGD with l_2-penalty

$$\theta_t = \theta_{t-1} - \eta\left(\mathbf{g}_t + 2\lambda\theta_{t-1}\right).$$

- This is also known as weight decay as the norm of θ decays rapidly due to the l_2-penalty.

AdamW: Adam with Decoupled Weight Decay Regularization

- Standard l_2 regularization is not equivalent to weight decay in Adam.
- AdamW explicitly decouples weight decay from the gradient update:

$$\theta_t = \theta_{t-1} - 2\lambda\theta_{t-1} - \frac{\eta_t \widehat{\mathbf{v}}_t}{\sqrt{\widehat{\mathbf{s}}_t} + \delta \cdot \mathbf{1}}$$

- AdamW typically achieves better performance across various language modeling tasks.

```
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)
```

Batch normalization (Ioffe and Szegedy, 2015)

- Motivation: solve a covariate shift problem (the input features (covariates) of the training data and testing data are different).

Batch normalization

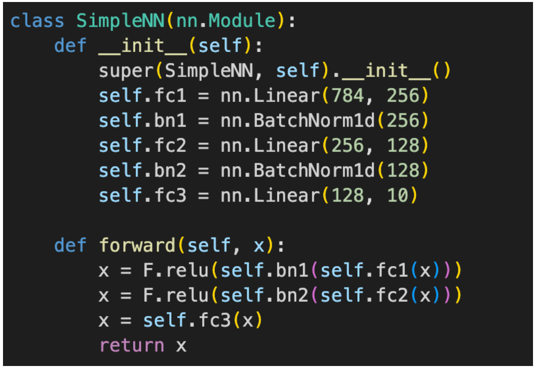
- For each input feature $\mathbf{x}$, let $\mathbf{h}^{l-1}(\mathbf{x}) = \sigma(\mathbf{W}^{l-1}\sigma(\cdots\sigma(\mathbf{W}^{[1]}\mathbf{x}+\mathbf{b}^1)) + \mathbf{b}^{l-1}) \in \mathbb{R}^{d^{l-1}}$ be the output of the $(l-1)$-th layer.
- For a minibatch $\mathcal{B}_t$ of size b, denote $\mathbf{a}_i = \mathbf{W}^l\mathbf{h}^{l-1}(\mathbf{x}_i) + \mathbf{b}^l$ for any $i \in \mathcal{B}_t$.
- Mean: $\boldsymbol{\mu} = \frac{1}{b}\sum_{i\in\mathcal{B}_t}\mathbf{a}_i$; Variance: $\boldsymbol{\sigma}^2 = \frac{1}{b}\sum_{i\in\mathcal{B}_t}(\mathbf{a}_i - \boldsymbol{\mu}) \odot (\mathbf{a}_i - \boldsymbol{\mu})$
- Normalization:
$$\widetilde{\mathbf{a}}_i = \frac{\mathbf{a}_i - \boldsymbol{\mu}}{\sqrt{\boldsymbol{\sigma}^2 + \delta}},$$
$\delta > 0$ is a small number to prevent 0 values.

Batch normalization

- Introduce two more (trainable) parameter vectors γ, β to allow for heterogeneity.
- For each $i \in \mathcal{B}_t$, output: $\boldsymbol{\gamma} \odot \widetilde{\mathbf{a}}_i + \boldsymbol{\beta}$
- Activation:

$$\sigma(\boldsymbol{\gamma} \odot \widetilde{\mathbf{a}}_i + \boldsymbol{\beta})$$

Batch normalization using pytorch



```python
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.fc1(x)))
        x = F.relu(self.bn2(self.fc2(x)))
        x = self.fc3(x)
        return x
```

Layer normalization

- Batch normalization (BN) depends on the mini-batch size.
- It is not obvious how to use BN in recurrent neural networks (RNN).
- RNN is important to model language tasks (next week).

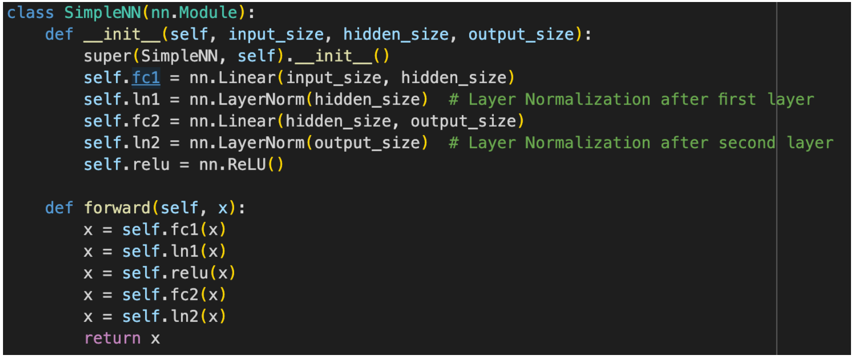
Layer normalization

- Batch normalization (BN) depends on the mini-batch size.
- It is not obvious how to use BN in recurrent neural networks (RNN).
- RNN is important to model language tasks (next week).

Layer normalization

- $\mathbf{a}_i = \mathbf{W}^l \mathbf{h}^{l-1}(\mathbf{x}_i) + \mathbf{b}^l \in \mathbb{R}^{d^l} =: (a_{i1}, \cdots, a_{id^l})$.
- Mean: $\mu = \frac{1}{d^l} \sum_{j=1}^{d^l} a_{ij}$; Variance: $\sigma^2 = \frac{1}{d^l} \sum_{j=1}^{d^l} (a_{ij} - \mu)^2$

Layer normalization using pytorch



```python
class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.ln1 = nn.LayerNorm(hidden_size)  # Layer Normalization after first layer
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.ln2 = nn.LayerNorm(output_size)  # Layer Normalization after second layer
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.ln1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.ln2(x)
        return x
```

Code

```
nn.Conv2d(1, 6, kernel_size=5, padding=2)
```

- Input channel size: 1
- Output channel size:6
- Kernel size:5
- Padding:2

Refers to Dive into Deep Learning