Deep Learning

Sequence regression (RNNs), stability and robustness

Lessons: **Kevin Scaman**
TPs: Paul Lerner

Class overview

Lessons

1. Introduction, simple architectures (MLPs) and autodiff 09/02
2. Training pipeline, optimization and image analysis (CNNs) 16/02
3. **Sequence regression (RNNs), stability and robustness** 08/03
4. Generative models in vision and text (Transformers, GANs) 15/03

Sequence prediction and classification

Text sequences

- Text auto-completion
- Sentiment analysis

Audio sequences

- Speech to text
- Music generation

Time-series forecasting

- Market price prediction
- Weather forecast

Standard approaches

Data: Sequences of the form $(x_1, \ldots, x_t)$. **Objective:** guess next iterate x_{t+1}.

Standard approaches

Data: Sequences of the form $(x_1, \ldots, x_t)$. **Objective:** guess next iterate x_{t+1}.

Classical ML models

- **Hidden Markov Models:** Probabilistic model where current value is drawn according to a distribution dependent on a hidden state.
- **Auto-regressive models:** Linear relationship between current and previous iterates.

Standard approaches

Data: Sequences of the form $(x_1, \ldots, x_t)$. **Objective:** guess next iterate x_{t+1}.

Classical ML models

- **Hidden Markov Models:** Probabilistic model where current value is drawn according to a distribution dependent on a hidden state.
- **Auto-regressive models:** Linear relationship between current and previous iterates.

Convolutional Neural Networks

- We can integrate the temporal dimension with a **1d convolution**.
- Standard architecture: **WaveNet** (Van den Oord et al., 2016)

Standard approaches

Data: Sequences of the form $(x_1, \ldots, x_t)$. **Objective:** guess next iterate x_{t+1}.

Classical ML models

- **Hidden Markov Models:** Probabilistic model where current value is drawn according to a distribution dependent on a hidden state.
- **Auto-regressive models:** Linear relationship between current and previous iterates.

Convolutional Neural Networks

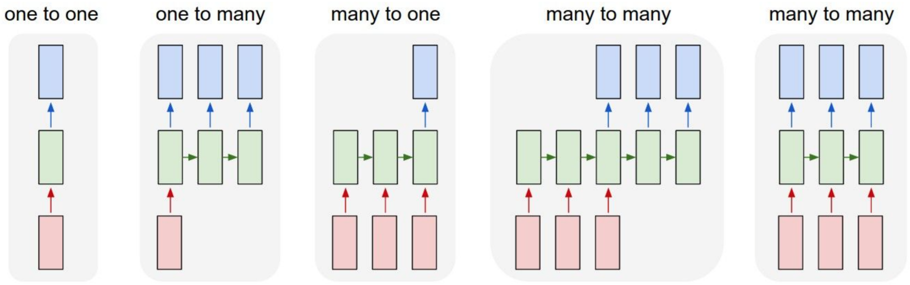
- We can integrate the temporal dimension with a **1d convolution**.
- Standard architecture: **WaveNet** (Van den Oord et al., 2016)

Transformers

- Based on a selection procedure using **attention** modules (see in next class).
- Current **state-of-the-art** for natural language processing.

Recurrent Neural Networks

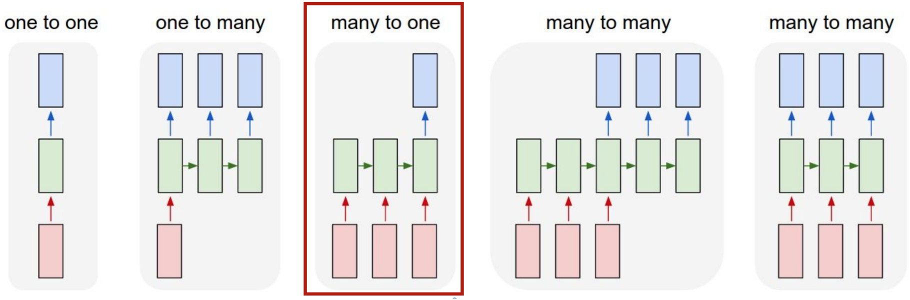
- Several variants



source: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

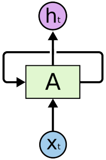
Recurrent Neural Networks

- Today



source: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Recurrent Neural Networks



Causality & short-term dependency

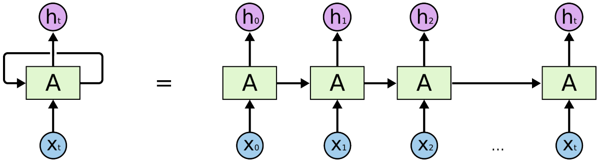
We process a sequence of vectors x_t by applying a **recurrence formula** at every time step:

$$h_t = f_W(h_{t-1}, x_t)$$

- h_{t-1} = previous state, h_t = current state
- f_W = some function with parameters W
- x_t = input column vector at time step t

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Recurrent Neural Networks



Usual implementation

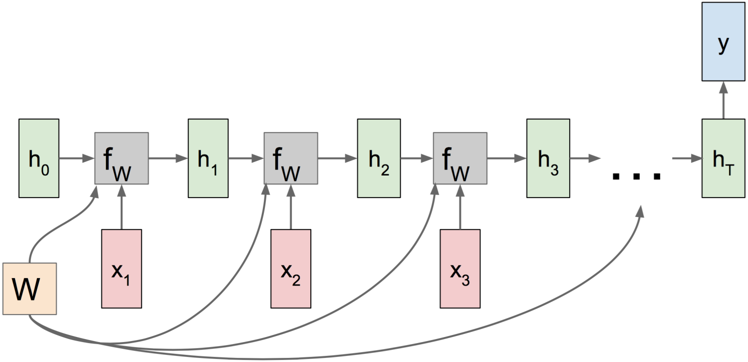
- Typically (note the use of the $\tanh$ non-linearity):

$$h_t = \tanh(W_{hh}\, h_{t-1} + W_{xh}\, x_t)$$

- Output: $y_t = W_{hy}\, h_t$ or $y_t = \mathrm{softmax}(W_{hy}\, h_t)$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

RNN computational graphs



Backpropagation through time

source: J. Johnson

A simple binary sequence classification problem

- Can you guess the task?

Sequence	Class
[1, 1, -1, -1, 1, -1]	1
[1, -1, 1, -1]	1
[1, -1, 1, 1, -1, 1, -1, -1]	1
[1, 1, -1, -1, -1, 1, -1, 1]	0
[1, -1, -1, 1, 1, -1]	0
[1, -1, -1, 1]	0

A simple binary sequence classification problem

- Can you guess the task?

Sequence	Class
[1, 1, -1, -1, 1, -1] = (())()	1
[1, -1, 1, -1] = ()()	1
[1, -1, 1, 1, -1, 1, -1, -1] = ()(()())	1
[1, 1, -1, -1, -1, 1, -1, 1] = (()))()(	0
[1, -1, -1, 1, 1, -1] = ())(()	0
[1, -1, -1, 1] = ())(	0

A simple binary sequence classification problem

- Can you guess the task?

Sequence	Class
[1, 1, -1, -1, 1, -1] = (())()	1
[1, -1, 1, -1] = ()()	1
[1, -1, 1, 1, -1, 1, -1, -1] = ()(()())	1
[1, 1, -1, -1, -1, 1, -1, 1] = (()))()(	0
[1, -1, -1, 1, 1, -1] = ())(()	0
[1, -1, -1, 1] = ())(	0

- How would you solve this task?

A (less) simple binary sequence classification problem

- We will make it a bit more complicated with **colored parenthesis**, example with 10 colors.
- **Rule:** Opening parenthesis $i \in [0, 4]$ with corresponding closing parenthesis $j \in [5, 9]$ such that $i + j = 9$.

Sequence	Class
[2, 0, 9, 7, 0, 9] = (())()	1
[1, 8, 3, 6] = ()()	1
[0, 9, 2, 4, 5, 2, 7, 7] = ()(()())	1
[0, 2, 7, 9, 7, 2, 7, 3] = (()))()(	0
[1, 8, 9, 0, 1, 9] = ())(()	0
[1, 8, 7, 1] = ())(	0

A (less) simple binary sequence classification problem

- We will make it a bit more complicated with **colored parenthesis**, example with 10 colors.
- **Rule:** Opening parenthesis $i \in [0, 4]$ with corresponding closing parenthesis $j \in [5, 9]$ such that $i + j = 9$.

Sequence	Class
[2, 0, 9, 7, 0, 9] = (())()	1
[1, 8, 3, 6] = ()()	1
[0, 9, 2, 4, 5, 2, 7, 7] = ()(()())	1
[0, 2, 7, 9, 7, 2, 7, 3] = (()))()(	0
[1, 8, 9, 0, 1, 9] = ())(()	0
[1, 8, 7, 1] = ())(	0

- How would you solve this task?

Elman network (1990)

First implementation of RNNs, simple ReLU activation and linear output.

- **Initial hidden state:** $h_0 = 0$
- **Update:** $h_t = \mathrm{ReLU}(W_{xh}\, x_t + W_{hh}\, h_{t-1} + b_h)$
- **Final prediction:** $y_T = W_{hy}\, h_T + b_y$

Elman network (1990)

First implementation of RNNs, simple ReLU activation and linear output.

- **Initial hidden state:** $h_0 = 0$
- **Update:** $h_t = \mathrm{ReLU}(W_{xh}\, x_t + W_{hh}\, h_{t-1} + b_h)$
- **Final prediction:** $y_T = W_{hy}\, h_T + b_y$

```
class RecNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, dim_output):
        super(RecNet, self).__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, x):
        h = x.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(x.size(0)):
            h = torch.relu(self.fc_x2h(x[t,:]) + self.fc_h2h(h))
        return self.fc_h2y(h)
```

Training

- We encode the symbol at time t as a one-hot vector x_t
- To simplify the processing of variable-length sequences, we are processing samples (i.e. sequences) one at a time. **We do not consider batches**.

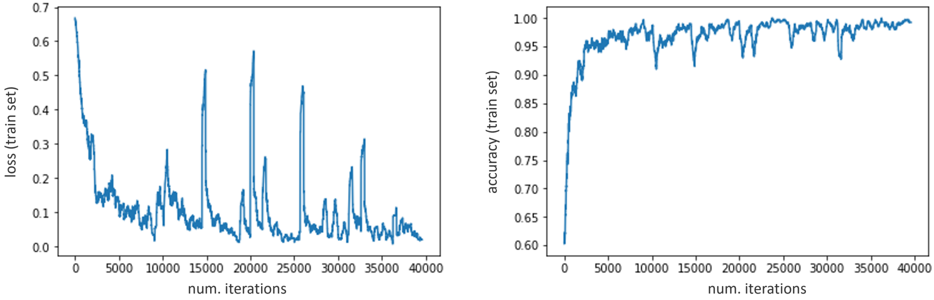
```
RNN = RecNet(dim_input = nb_symbol, dim_recurrent=50, dim_output=2)

cross_entropy = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(RNN.parameters(), lr=learning_rate)

for k in range(nb_train):
    x,l = generator.generate_input()
    y = RNN(x)
    loss = cross_entropy(y,l)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Results



- Loss decreases and fraction of correct classification increases but did our network learn?

Gating

Main idea

- Gates are a way to optionally let information through.
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!".

Gating

Main idea

- Gates are a way to optionally let information through.
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!".

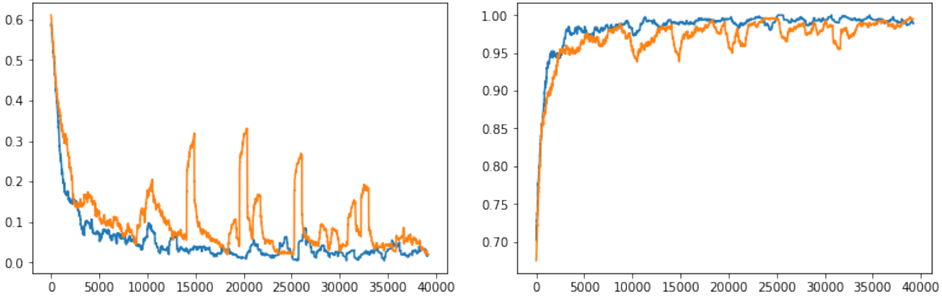
- **Recurrence relation:** $\overline{h}_t = \mathrm{ReLU}(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$
- **Forget gate:** $z_t = \mathrm{sigm}(W_{xz}x_t + W_{hz}h_{t-1} + b_z)$
- **Hidden state:** $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t$

Gated RNN

```
class RecNetGating(nn.Module):
    def __init__(self, dim_input=10, dim_recurrent=50, dim_output=2):
        super(RecNetGating, self).__init__()
        self.fc_x2h = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2h = nn.Linear(dim_recurrent, dim_recurrent, bias = False)
        self.fc_x2z = nn.Linear(dim_input, dim_recurrent)
        self.fc_h2z = nn.Linear(dim_recurrent,dim_recurrent, bias = False)
        self.fc_h2y = nn.Linear(dim_recurrent, dim_output)

    def forward(self, x):
        h = x.new_zeros(1, self.fc_h2y.weight.size(1))
        for t in range(x.size(0)):
            z = torch.sigmoid(self.fc_x2z(x[t,:])+self.fc_h2z(h))
            hb = torch.relu(self.fc_x2h(x[t,:]) + self.fc_h2h(h))
            h = z * h + (1-z) * hb
        return self.fc_h2y(h)
```

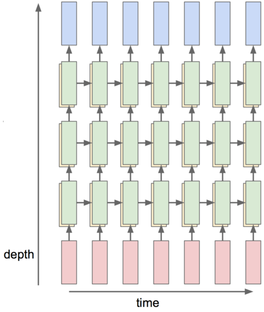
Results



- Orange = previous RNN.
- Blue = Gated RNN.
- Is there a benefit with gating?

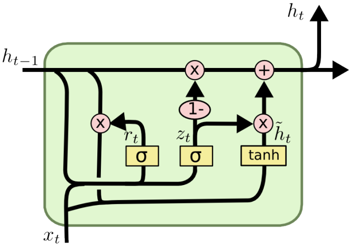
LSTM, GRU and multi-layer RNNs

- More parameters than Elman networks (simple RNN).
- Mitigates **vanishing gradient** problem through **gating**.
- Widely used and SOTA in many sequence learning problems.



GRU: Gated Recurrent Unit (Cho et al., 2014)

- **Recurrence relation:** $\overline{h}_t = \tanh(W_{xh}\, x_t + W_{hh}\,(r_t \odot h_{t-1}) + b_h)$
- **Forget gate:** $z_t = \mathrm{sigm}(W_{xz} x_t + W_{hz} h_{t-1} + b_z)$
- **Reset gate:** $r_t = \mathrm{sigm}(W_{xr}\, x_t + W_{hr}\, h_{t-1} + b_r)$
- **Hidden state:** $h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t$



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

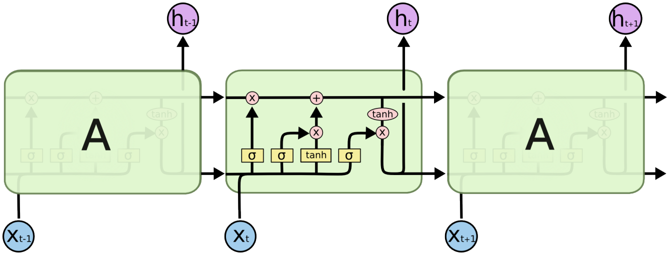
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

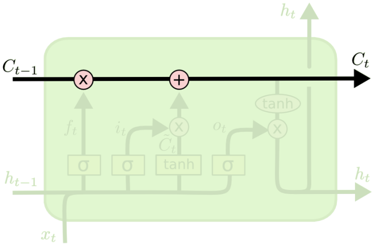
LSTM: Long Short-Term Memory (Hochreiter and Schmidhuber, 1997)



source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Inside LSTMs

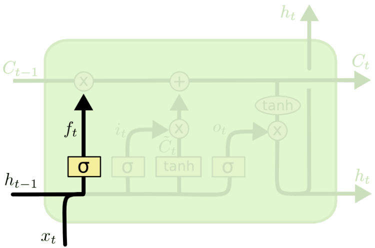
- Cell state



source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Inside LSTMs

- Forget gate layer

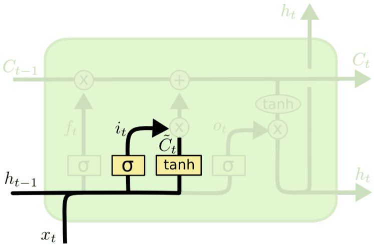


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Inside LSTMs

- Input gate layer



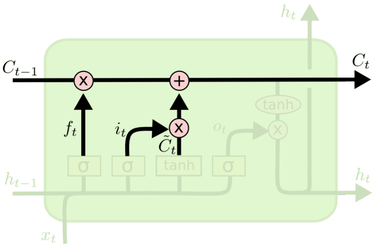
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Inside LSTMs

- Update cell state

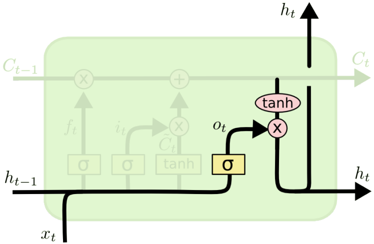


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Inside LSTMs

- Output gate



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

source: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

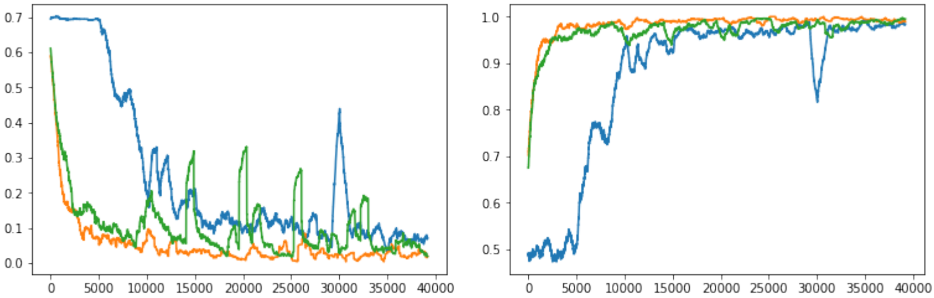
LSTMs in PyTorch

```
class LSTMNet(nn.Module):
    def __init__(self, dim_input, dim_recurrent, num_layers, dim_output):
        super(LSTMNet, self).__init__()
        self.lstm = nn.LSTM(input_size = dim_input,
                            hidden_size = dim_recurrent,
                            num_layers = num_layers)
        self.fc_o2y = nn.Linear(dim_recurrent,dim_output)

    def forward(self, x):
        x = x.unsqueeze(1)
        output, _ = self.lstm(x)
        # only last layer, shape (seq. len., bs, dim_recurrent)
        # drop the batch index
        output = output.squeeze(1)
        # keep only the last hidden variable
        output = output.narrow(0, output.size(0)-1,1)
        # shape (1, dim_recurrent)
        return self.fc_o2y(F.relu(output))
```

Note: the prediction is done from the hidden state, hence also called the output state.

Results



- Green = Elman RNN.
- Orange = Gated RNN.
- Blue = LSTM.
- Is there a benefit with LSTM?

Common wisdom in 2015

- Josefowicz et al. (2015) conducted an extensive exploration of different recurrent architectures, they wrote:

  "We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably outperforms the LSTM. Though there were architectures that outperformed the LSTM on some problems, **we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions***."*

Common wisdom in 2015

- Josefowicz et al. (2015) conducted an extensive exploration of different recurrent architectures, they wrote:

  "We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably outperforms the LSTM. Though there were architectures that outperformed the LSTM on some problems, **we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions***."*

- Now let see if the LSTM is performing better on our task of checking for **balanced parentheses**!

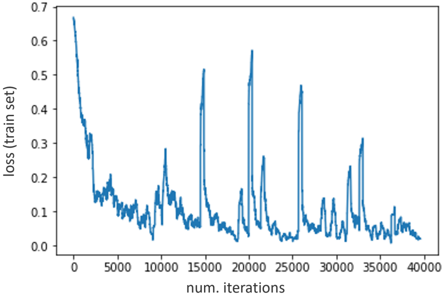
Stability during training

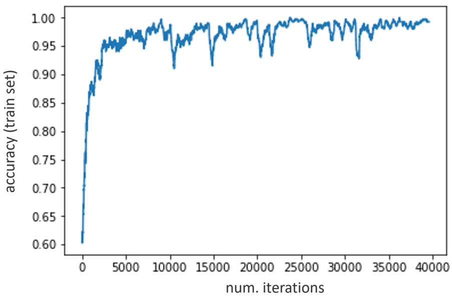
Weights initialization, gradient vanishing and explosion

Stability during training

Example with simple RNNs (Elman networks, no gating mechanisms)

- The gradients are sometimes **very large**.
- This leads to a large **drop in accuracy**.
- Results are **quite random**, final performance depends on initialization.





Gradient vanishing and explosion

Breaking gradient descent

- If θ_t are the iterates of the parameters learned using stochastic gradient descent on minibatches $(x_{t,i}, y_{t,i})_{i\in[\![1,K]\!]}$ at time t, then we have

$$\theta_{t+1} = \theta_t - \frac{\eta}{K}\sum_i \nabla\mathcal{L}_{x_{t,i},y_{t,i}}(\theta)\,,$$

  where $\mathcal{L}_{x,y}(\theta) = \ell(g_\theta(x), y)$.
- **Gradient vanishing:** When the gradients $\nabla\mathcal{L}_{x_{t,i},y_{t,i}}(\theta)$ are very small compared to θ_t, the iteration does not modify the parameters.
- **Gradient explosion:** When the gradients $\nabla\mathcal{L}_{x_{t,i},y_{t,i}}(\theta)$ are very large compared to θ_t, the iteration will push the parameters to extreme values.

Gradient vanishing and explosion

Why is it a problem for deep learning?

- By chain rule, the gradient tends to multiply along the layers.
- Example: If $g^{(L)}(x) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(x)$ where $f^{(L)} : \mathbb{R} \to \mathbb{R}$, then

$$g^{(L)'}(x) = \prod_{l=1}^{L} f^{(l)'}(g^{(l-1)}(x))$$

- If $f^{(l)'}(g^{(l-1)}(x)) \approx c$, then $g^{(L)'}(x) \approx c^L$.
- **Exponentially small** w.r.t. L if $c < 1$ (gradient vanishing).
- **Exponentially large** w.r.t. L if $c > 1$ (gradient explosion).

Mitigation techniques: how to avoid this?

Gradient clipping

- `torch.nn.utils.clip_grad_norm_(model.parameters(), threshold)`
- **Pros:** Easiest method, just limits the gradient norm to a fixed value.
- **Cons:** Only for gradient explosion, adds an extra hyper-parameter.

Mitigation techniques: how to avoid this?

Gradient clipping

- `torch.nn.utils.clip_grad_norm_(model.parameters(), threshold)`
- **Pros:** Easiest method, just limits the gradient norm to a fixed value.
- **Cons:** Only for gradient explosion, adds an extra hyper-parameter.

Architecture changes

- Gates in RNNs, residuals in CNNs, dropout, batch normalization, ...
- **Pros:** More principled, usually leads to better performance.
- **Cons:** Requires to change the network architecture, application dependent.

Mitigation techniques: how to avoid this?

Gradient clipping

- `torch.nn.utils.clip_grad_norm_(model.parameters(), threshold)`
- **Pros:** Easiest method, just limits the gradient norm to a fixed value.
- **Cons:** Only for gradient explosion, adds an extra hyper-parameter.

Architecture changes

- Gates in RNNs, residuals in CNNs, dropout, batch normalization, ...
- **Pros:** More principled, usually leads to better performance.
- **Cons:** Requires to change the network architecture, application dependent.

Weight initialization

- Automatically implemented, but can have an **large impact** on performance

Weights initialization

Ideal initialization scheme

- The better the model is at initialization, the more changes we have of find good weights.
- We would like to have values that are reasonable, $\forall i \in [\![1, d^{(L)}]\!]$, $|g_\theta(x)_i| \approx 1$.
- We would like to have gradients that are neither too large nor too small

$$\forall i \in [\![1, p]\!], \qquad |\nabla \mathcal{L}_{x,y}(\theta)_i| \approx 1$$

Weights initialization

Ideal initialization scheme

- The better the model is at initialization, the more changes we have of find good weights.
- We would like to have values that are reasonable, $\forall i \in [\![1, d^{(L)}]\!]$, $|g_\theta(x)_i| \approx 1$.
- We would like to have gradients that are neither too large nor too small

$$\forall i \in [\![1, p]\!], \qquad |\nabla \mathcal{L}_{x,y}(\theta)_i| \approx 1$$

Simple solution

- Set $b^{(l)} = 0$ and sample the weights $W_{ij}^{(l)} \sim \mathcal{P}$ i.i.d. with expectation 0 and variance $V^{(l)}$.
- Choose $V^{(l)}$ so that the variance is constant across layers.

Weights initialization

Ideal initialization scheme

- The better the model is at initialization, the more changes we have of find good weights.
- We would like to have values that are reasonable, $\forall i \in [\![1, d^{(L)}]\!]$, $|g_\theta(x)_i| \approx 1$.
- We would like to have gradients that are neither too large nor too small

$$\forall i \in [\![1, p]\!], \qquad |\nabla \mathcal{L}_{x,y}(\theta)_i| \approx 1$$

Simple solution

- Set $b^{(l)} = 0$ and sample the weights $W_{ij}^{(l)} \sim \mathcal{P}$ i.i.d. with expectation 0 and variance $V^{(l)}$.
- Choose $V^{(l)}$ so that the variance is constant across layers.
- Technical assumptions:
  - The probability distribution is symmetric w.r.t. 0 and $\mathcal{P}(\{0\}) = 0$.
  - The activation function is ReLU $\sigma(x) = \max\{0, x\}$.

Derivation of optimal weight variance

Preliminary results

- Let $x \in \mathbb{R}^{d^{(0)}}$ a fixed input and, $\forall l \in [\![1, L]\!]$, $X^{(l)} = g_\theta^{(2l-1)}(x)$.
- For any $l \in [\![1, L]\!]$, the variables $(X_i^{(l)})_{i \in [\![1, d^{(2l-1)}]\!]}$ are identically distributed.
- The distribution of $X_i^{(l)}$ is symmetric w.r.t. 0 (and thus $\mathbb{E}(X_j^{(l)}) = 0$).

Derivation of optimal weight variance

Preliminary results

- Let $x \in \mathbb{R}^{d^{(0)}}$ a fixed input and, $\forall l \in [\![1, L]\!]$, $X^{(l)} = g_\theta^{(2l-1)}(x)$.
- For any $l \in [\![1, L]\!]$, the variables $(X_i^{(l)})_{i \in [\![1, d^{(2l-1)}]\!]}$ are identically distributed.
- The distribution of $X_i^{(l)}$ is symmetric w.r.t. 0 (and thus $\mathbb{E}(X_j^{(l)}) = 0$).

Proof.

- The proof follows a simple recurrence:

Derivation of optimal weight variance

Preliminary results

- Let $x \in \mathbb{R}^{d^{(0)}}$ a fixed input and, $\forall l \in [\![1, L]\!]$, $X^{(l)} = g_\theta^{(2l-1)}(x)$.
- For any $l \in [\![1, L]\!]$, the variables $(X_i^{(l)})_{i \in [\![1, d^{(2l-1)}]\!]}$ are identically distributed.
- The distribution of $X_i^{(l)}$ is symmetric w.r.t. 0 (and thus $\mathbb{E}(X_j^{(l)}) = 0$).

Proof.

- The proof follows a simple recurrence:
- Initialization: $X_i^{(1)} = \sum_j W_{ij}^{(1)} x_j$ is identically distributed and symmetric.

Derivation of optimal weight variance

Preliminary results

- Let $x \in \mathbb{R}^{d^{(0)}}$ a fixed input and, $\forall l \in [\![1, L]\!]$, $X^{(l)} = g_\theta^{(2l-1)}(x)$.
- For any $l \in [\![1, L]\!]$, the variables $(X_i^{(l)})_{i \in [\![1, d^{(2l-1)}]\!]}$ are identically distributed.
- The distribution of $X_i^{(l)}$ is symmetric w.r.t. 0 (and thus $\mathbb{E}(X_j^{(l)}) = 0$).

Proof.

- The proof follows a simple recurrence:
- Initialization: $X_i^{(1)} = \sum_j W_{ij}^{(1)} x_j$ is identically distributed and symmetric.
- If the properties are verified for l, then $X_i^{(l)} = \sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})$, which is identically distributed and symmetric.

□

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$\mathrm{var}(X_i^{(l)}) \quad = \quad \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)}))$$

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$\begin{aligned} \mathrm{var}(X_i^{(l)}) &= \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\ &= \textstyle\sum_j \mathrm{var}(W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \end{aligned}$$

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$
\begin{aligned}
\mathrm{var}(X_i^{(l)}) &= \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= \textstyle\sum_j \mathrm{var}(W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= d^{(l-1)} \ \mathrm{var}(W_{ij}^{(l)})\, \mathbb{E}(\sigma(X_j^{(l-1)})^2)
\end{aligned}
$$

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$
\begin{aligned}
\mathrm{var}(X_i^{(l)}) &= \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= \textstyle\sum_j \mathrm{var}(W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= d^{(l-1)} \ \mathrm{var}(W_{ij}^{(l)})\, \mathbb{E}(\sigma(X_j^{(l-1)})^2) \\
&= d^{(l-1)}\, V^{(l)}\, \mathbb{E}({X_j^{(l-1)}}^2 \mathbb{1}\{X_j^{(l-1)} > 0\})
\end{aligned}
$$

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$
\begin{aligned}
\mathrm{var}(X_i^{(l)}) &= \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= \textstyle\sum_j \mathrm{var}(W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= d^{(l-1)}\ \mathrm{var}(W_{ij}^{(l)})\, \mathbb{E}(\sigma(X_j^{(l-1)})^2) \\
&= d^{(l-1)}\, V^{(l)}\, \mathbb{E}({X_j^{(l-1)}}^2 \mathbb{1}\{X_j^{(l-1)} > 0\}) \\
&= d^{(l-1)}\, V^{(l)}\ \mathrm{var}(X_j^{(l-1)})/2
\end{aligned}
$$

Variance of the output value

Variance of the intermediate outputs

- For any $l \in [\![2, L]\!]$ and $i \in [\![1, d^{(2l-1)}]\!]$, we have

$$
\begin{aligned}
\mathrm{var}(X_i^{(l)}) &= \mathrm{var}(\textstyle\sum_j W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= \textstyle\sum_j \mathrm{var}(W_{ij}^{(l)} \sigma(X_j^{(l-1)})) \\
&= d^{(l-1)}\ \mathrm{var}(W_{ij}^{(l)})\, \mathbb{E}(\sigma(X_j^{(l-1)})^2) \\
&= d^{(l-1)}\, V^{(l)}\, \mathbb{E}({X_j^{(l-1)}}^2 \mathbb{1}\{X_j^{(l-1)} > 0\}) \\
&= d^{(l-1)}\, V^{(l)}\ \mathrm{var}(X_j^{(l-1)})/2
\end{aligned}
$$

- Hence, the **variance is constant across layers** if $V^{(l)} = 2/d^{(l-1)}$, and

$$\mathrm{var}(g_\theta(x)_i) = 2\|x\|_2^2/d^{(0)}$$

Kaiming initialization (Kaiming He et.al., 2015)

Gaussian weights

Our assumptions are satisfied if we use Gaussian weights $W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{d^{(l-1)}}\right)$.

Uniform weights

If we take uniform weights $W_{ij}^{(l)} \sim \mathcal{U}([-r^{(l)}, r^{(l)}])$, then $V^{(l)} = r^2/3$ and

$$r^{(l)} = \sqrt{\frac{6}{d^{(l-1)}}}$$

Variance of the gradient

Variance propagation during backprop

- Same analysis for backprop, but in **reverse**.
- This gives an optimal variance $V^{(l)} = 2/d^{(l)}$.

Variance of the gradient

Variance propagation during backprop

- Same analysis for backprop, but in **reverse**.
- This gives an optimal variance $V^{(l)} = 2/d^{(l)}$.
- In order to have both the variances of gradients and of values constant, we thus need $V^{(l)} = 2/d^{(l)}$ and $V^{(l)} = 2/d^{(l-1)}$...

Variance of the gradient

Variance propagation during backprop

- Same analysis for backprop, but in **reverse**.
- This gives an optimal variance $V^{(l)} = 2/d^{(l)}$.
- In order to have both the variances of gradients and of values constant, we thus need $V^{(l)} = 2/d^{(l)}$ and $V^{(l)} = 2/d^{(l-1)}$...
- A reasonable heuristic consists in taking the average: $V^{(l)} = \frac{4}{d^{(l)}+d^{(l-1)}}$.

Variance of the gradient

Variance propagation during backprop

- Same analysis for backprop, but in **reverse**.
- This gives an optimal variance $V^{(l)} = 2/d^{(l)}$.
- In order to have both the variances of gradients and of values constant, we thus need $V^{(l)} = 2/d^{(l)}$ and $V^{(l)} = 2/d^{(l-1)}$...
- A reasonable heuristic consists in taking the average: $V^{(l)} = \frac{4}{d^{(l)}+d^{(l-1)}}$.

Xavier initialization (Xavier Glorot & Yoshua Bengio, 2010)

Let $c > 0$ be a hyper-parameter. The weights are initialized using the heuristic

$$W_{ij}^{(l)} \sim \mathcal{U}([-r^{(l)}, r^{(l)}]) \qquad \text{and} \qquad r^{(l)} = \sqrt{\frac{6c^2}{d^{(l)} + d^{(l-1)}}}$$

Impact of initialization in practice: `https://www.deeplearning.ai/ai-notes/initialization/index.html`

Batch normalization

Idea

- Normalize the input of each layer by **removing mean and dividing by std**.
- Also uses a **learnable affine map**.

Batch normalization

Idea

- Normalize the input of each layer by **removing mean and dividing by std**.
- Also uses a **learnable affine map**.

Definition

- If $(x_i)_i$ is a batch of b inputs (to the layer), then the output is:

$$y_i = \frac{x_i - E}{\sqrt{V + \varepsilon}} \cdot \gamma + \beta$$

where $E = \frac{1}{b} \sum_i x_i$ and $V = \frac{1}{b} \sum_i (x_i - E)^2$ (coord.-wise), γ and β are learnable vectors.

Batch normalization

The output depends on the whole batch, not just single inputs!

Train and eval

- The behavior of batch norm is different between training and evaluation (e.g. `model.train()` and `model.eval()` in Pytorch).
- At evaluation, the model uses a (moving) average of **all training batches**.
- Stores E and V for each training batch, and then computes

$$(1-\rho)\sum_t \rho^t E_t \quad \text{and} \quad (1-\rho)\sum_t \rho^t V_t$$

where (typically) $\rho = 0.9$.

Recap

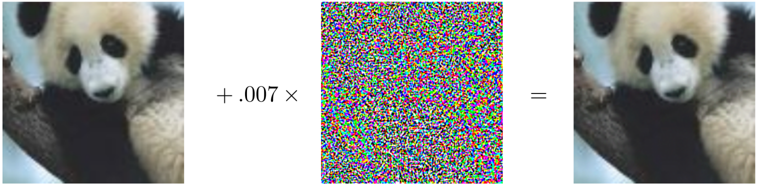
- Gradient **vanishing** and **explosion** can happen during training of **deep** NNs.
- **Gradient clipping**, **batch normalization**, **regularisation** and proper **weight initialization** can help stabilize training.
- The variance of the weights at initialization should be **inversely proportional to the layer width**.

Robustness and adversarial attacks

Confusing a neural network with noise

Adversarial attacks

- Can a small (invisible) noise change the prediction of a vision model?
- Vision models are robust to random input noise.
- Vision models are **extremely fragile** to **well-crafted** input noise.



$\boldsymbol{x}$ + .007 × $\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$ = $\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$

"panda" 57.7% confidence — "nematode" 8.2% confidence — "gibbon" 99.3 % confidence

source: Explaining and Harnessing Adversarial Examples, Goodfellow et al, ICLR 2015.

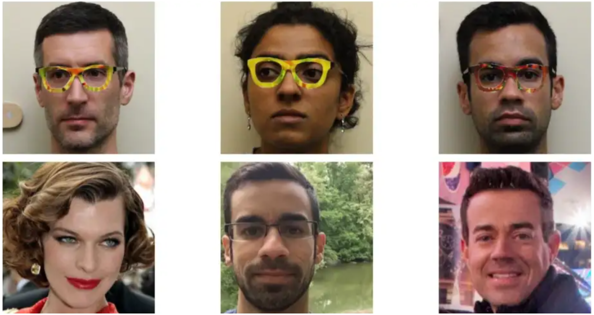
Adversarial attacks

- Can a small (invisible) noise change the prediction of a vision model?
- Vision models are robust to random input noise.
- Vision models are **extremely fragile** to **well-crafted** input noise.

source: Robust Physical-World Attacks on Deep Learning Visual Classification, Eykholt et al, CVPR 2018.

Adversarial attacks

- Can a small (invisible) noise change the prediction of a vision model?
- Vision models are robust to random input noise.
- Vision models are **extremely fragile** to **well-crafted** input noise.



source: Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition, Sharif et.al., CCS 2016.

Adversarial attacks: examples

Fast gradient sign method (Goodfellow et.al., 2014)

- **Idea:** Take one gradient step in the direction that **maximizes the loss**.
- To control the maximum pixel noise, use the coordinates' sign instead of value.
- **Limitations:** Destroys performance, but cannot target a specific class.

$$x^{\text{att}} = x^{\text{true}} + \varepsilon \, \mathsf{sign}(\nabla_x \mathcal{L}(\theta, x^{\text{true}}, y^{\text{true}}))$$

Adversarial attacks: examples

Fast gradient sign method (Goodfellow et.al., 2014)

- **Idea:** Take one gradient step in the direction that **maximizes the loss**.
- To control the maximum pixel noise, use the coordinates' sign instead of value.
- **Limitations:** Destroys performance, but cannot target a specific class.

$$x^{\text{att}} = x^{\text{true}} + \varepsilon\,\mathsf{sign}(\nabla_x \mathcal{L}(\theta, x^{\text{true}}, y^{\text{true}}))$$

Iterative Target Class Method (Kurakin et.al., 2016)

- **Idea:** Perform gradient descent on the loss with **labels swaped**.
- To control the maximum pixel noise, project on a ball of radius ε around x.
- **Limitations:** Requires to know the model weights (white box setting).

$$x^{\text{att}}_{k+1} = \mathsf{Clamp}_{x^{\text{true}},\varepsilon}\,(x^{\text{att}}_k + \varepsilon\,\mathsf{sign}(\nabla_x \mathcal{L}(\theta, x^{\text{att}}_k, y^{\text{att}})))$$

Beyond the white box setting

White-box attacks

- Use the knowledge of the model to create the perturbation.
- Gradient descent on a modified objective (classes swapped).

Beyond the white box setting

White-box attacks

- Use the knowledge of the model to create the perturbation.
- Gradient descent on a modified objective (classes swapped).

Black-box attacks

- Attacks without access the parameters of the model.
- Use a similar model, usually works relatively well.

Beyond the white box setting

White-box attacks

- Use the knowledge of the model to create the perturbation.
- Gradient descent on a modified objective (classes swapped).

Black-box attacks

- Attacks without access the parameters of the model.
- Use a similar model, usually works relatively well.

Defenses

- Augment the dataset with adversarial attacks (brute-force).
- Control the smoothness of the model (see next).

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Lipschitz continuity

- First order approximation: $g_\theta(x+\varepsilon) - g_\theta(x) = J_{g,x}(x,\theta)\varepsilon + o(\|\varepsilon\|)$.

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Lipschitz continuity

- First order approximation: $g_\theta(x+\varepsilon) - g_\theta(x) = J_{g,x}(x,\theta)\varepsilon + o(\|\varepsilon\|)$.
- Control on $\|J_{g_\theta}(x)\|_2 = \max_{u\neq 0} \frac{\|J_{g_\theta}(x)u\|_2}{\|u\|_2}$ (operator norm) leads to robustness.

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Lipschitz continuity

- First order approximation: $g_\theta(x+\varepsilon) - g_\theta(x) = J_{g,x}(x,\theta)\varepsilon + o(\|\varepsilon\|)$.
- Control on $\|J_{g_\theta}(x)\|_2 = \max_{u\neq 0} \frac{\|J_{g_\theta}(x)u\|_2}{\|u\|_2}$ (operator norm) leads to robustness.
- **Lipschitz constant:** $L_{g_\theta} = \sup_x \|J_{g_\theta}(x)\|_2$.

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Lipschitz continuity

- First order approximation: $g_\theta(x+\varepsilon) - g_\theta(x) = J_{g,x}(x,\theta)\varepsilon + o(\|\varepsilon\|)$.
- Control on $\|J_{g_\theta}(x)\|_2 = \max_{u\neq 0} \frac{\|J_{g_\theta}(x)u\|_2}{\|u\|_2}$ (operator norm) leads to robustness.
- **Lipschitz constant:** $L_{g_\theta} = \sup_x \|J_{g_\theta}(x)\|_2$.
- For piece-wise linear interpolation, Lipschitz constant is **smaller than target function**.

Robustness of neural networks

What makes a model robust?

- **Vital** for practical applications in **engineering** or **medicine**.
- If **black-box**, then trusting the model requires **hard constraints**.
- **Small input perturbation leads to small output perturbation**.

Lipschitz continuity

- First order approximation: $g_\theta(x+\varepsilon) - g_\theta(x) = J_{g,x}(x,\theta)\varepsilon + o(\|\varepsilon\|)$.
- Control on $\|J_{g_\theta}(x)\|_2 = \max_{u\neq 0} \frac{\|J_{g_\theta}(x)u\|_2}{\|u\|_2}$ (operator norm) leads to robustness.
- **Lipschitz constant:** $L_{g_\theta} = \sup_x \|J_{g_\theta}(x)\|_2$.
- For piece-wise linear interpolation, Lipschitz constant is **smaller than target function**.
- For neural networks: $L_{g_\theta} \leqslant \prod_l L_{f^{(l)}}$... can be exponential in number of layers!