Ceng 783 – Deep Learning

Week 5 – Convolutional Neural Networks (continued)

Fall 2019

Emre Akbas
Misc

- Project proposal feedback
- Hw #2 is due next Tuesday (Oct 29th).
- No lecture next week. Oct 29th is the Republic Day.
- Midterm exam on Nov 5th.
So far:

Neural network

... 

Linear and non-linear operations are stacked hierarchically.

Estimated label $\hat{y}$

Input $x$
So far:

Estimated label $\hat{y}$

Neural network

Matrix multiplication (FC), convolution

Linear and non-linear operations are stacked hierarchically. ReLU, sigmoid, pooling, etc.

Different connectivity types (fully, convolutional, local)

Input $x$
So far:

Estimated label $\hat{y}$

Neural network

Input $x$

Feedforward evaluation during testing/predicting
So far: training

\[
\text{Loss}(\hat{y}, y)
\]

\[
\hat{y} \rightarrow \text{Neural network} \rightarrow y
\]

Input $x$ \hspace{1cm} True label $y$
So far: training

\[ \text{Loss}(\hat{y}, y) \]

- Cross-entropy
- Hinge loss

\[ \hat{y} \]

- Softmax

\[ y \]

**Neural network**

**Input** \( x \)

**True label** \( y \)
So far: training

Loss($\hat{y}, y$)  Cross-entropy
Hinge loss

Gradient of loss w.r.t. parameters are computed using backpropagation.

Then, use a stochastic gradient descent method to minimize loss.
Today

Miscellaneous topics about ConvNets

• Loss/cost functions (cont’d)
  – Multiclass hinge
  – Derivation of cross-entropy as ML estimation

• Implementing backpropagation

• On the initialization of Nns

• Batch normalization

• Stochastic gradient descent variants
Loss/cost functions
Multiclass hinge loss

Training example \((x,y)\); neural network \(f()\)

\[ f : x \rightarrow C \text{ dimensional vector} \]

\[ \text{Hinge}(f(x), y) = \max \{0, 1 - (f_y(x) - \max_{c \neq y} f_c(x))\} \]
Cross-entropy

\[ L(\theta) = - \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\( y_i \) is a C-dimensional one-hot vector
\( q_i \) is the softmax of \( f(x) \)
Cross-entropy

\[ L(\theta) = - \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\( y_i \) is a C-dimensional one-hot vector
\( q_i \) is the softmax of \( f(x) \)

Q1: What does softmax do?
Cross-entropy

\[ L(\theta) = - \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\( y_i \) is a C-dimensional one-hot vector
\( q_i \) is the softmax of \( f(x) \)

Q1: What does softmax do?

- Normalize the raw scores output by the neural network
- Highlight the max score

\[ q_{ic} = \frac{e^{f_c(x_i)}}{\sum_k e^{f_k(x_i)}} \]
Cross-entropy

\[ L(\theta) = -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log q_{ic} \]

\(y_i\) is a C-dimensional one-hot vector
\(q_i\) is the softmax of f(x)

Q2: Where does cross-entropy come from?
Cross-entropy

Definition:

\[ H(p, q) = - \sum_x p(x) \log q(x) \]

Minimized when \( p=q \)
Cross-entropy

Comes from maximum-likelihood estimation

[Derivation on board]
Cross-entropy loss as maximum likelihood (ML) estimation

Given a dataset $S = \{\text{example}_i\}^N_{i=1}$, and a model,

$$P(S | \text{model}) = \text{?} \quad \text{or} \quad P(S ; \Theta) = \text{?}$$

model's parameters

$$\text{ML} : \quad P(S; \Theta) = \frac{1}{N} \prod_{i=1}^{N} p(\text{example}_i ; \Theta)$$

i.i.d. assumption

Let's derive cross-entropy: We have a supervised dataset:

$S = \{X, Y\}$ where $X$ matrix contains $N$ examples ($x_i$'s) and $Y$ contains the corresponding labels.

we want $\max_{\Theta} \quad P(Y | X ; \Theta)$

Use ML $\Rightarrow P(Y | X ; \Theta) = \frac{1}{N} \prod_{i=1}^{N} p(y_i | x_i ; \Theta)$

$y_i$ is a one-hot vector and the model output is a $C$-dimensional $C$-dim vector containing estimated class probabilities.
\[ p(y_i | x_i) = p(y_{i1} | x_i) \cdot p(y_{i2} | x_i) \cdot \ldots \cdot p(y_{ic} | x_i)^{y_{ic}} \]

\[ = \frac{1}{C} \prod_{c=1}^{C} p(y_{ic} | x_i)^{y_{ic}} \]

So ML is:

\[ \max_\theta \prod_{i=1}^{N} \prod_{c=1}^{C} p(y_{ic} | x_i)^{y_{ic}} \]

Take log:

\[ \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log p(y_{ic} | x_i) \]

Let multiply with -1 to make it a minimization problem:

\[ -\sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log p(y_{ic} | x_i) \]

This can be implemented using softmax normalization:

\[ p(y_{ic} | x_i) = \frac{e^{f_c(x_i)}}{\sum_k e^{f_k(x_i)}} \]

Where \( f_c(x) \) is model's score for \( x \) belonging to class \( c \).
On the implementation of backpropagation

The “modular” approach
A neural network is nothing but a composition of several linear and non-linear functions:

\[ y = f_k(f_{k-1}(\ldots f_1(x; \theta_1); \theta_{k-1}); \theta_k) \]

Given a specific architecture, i.e. composition, one can easily write the gradient w.r.t. parameters.

But a modular approach is desirable so that we don't have to derive the gradient again and again.

We can “compose” new architectures by simply connecting computing blocks.
A computing block:

Input $x$ → Function $f()$ → Output $o$

Params $w$

Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input: $\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$

Derivative of output w.r.t. parameters: $\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$
A computing block:

Input $x$ \hspace{1cm} \text{Function } f() \hspace{1cm} \text{Params } w \hspace{1cm} \text{Output } o

Forward pass: $o = f(x; w)$

Derivative of output w.r.t. input: 
$$\frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x}$$

Derivative of output w.r.t. parameters: 
$$\frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w}$$

Typically, $X$, $o$ and $w$ are vectors or matrices. Care has to be taken in computing the derivatives.
**Exercise:** try to work out all the details for a fully connected layer with D input nodes and K output nodes, receiving N examples.

Forward pass: \( o = f(x; w) \)

Derivative of output w.r.t. input: \( \frac{\partial o}{\partial x} = \frac{\partial f(x; w)}{\partial x} \)

Derivative of output w.r.t. parameters: \( \frac{\partial o}{\partial w} = \frac{\partial f(x; w)}{\partial w} \)
\[ X \times W \text{ where } W \text{ is } D\text{-by-}K \]

Derivative of output w.r.t. input:
\[ \frac{\partial o}{\partial X} = W \]

Derivative of output w.r.t. parameters:
\[ \frac{\partial o}{\partial W} = X \]
**Exercise:** do the same for a ReLU layer receiving N-by-K input

\[ X: N\text{-by-}K \rightarrow \text{Max}(0,x) \rightarrow o: N\text{-by-}K \]

Derivative of output w.r.t. input:  ?

Derivative of output w.r.t. parameters:  ?
Exercise: do the same for a ReLU layer receiving N-by-K input

Let's consider a ReLU layer with an input of size N-by-K and an output of size N-by-K. The layer takes the following form:

\[ \text{Max}(0, x) \]

where \( x \) is the input of size N-by-K. The output \( o \) is also of size N-by-K.

The derivative of the output with respect to the input is given by:

\[ \frac{\partial o}{\partial x_{ij}} = \begin{cases} 1 & \text{if } x_{ij} > 0 \\ 0 & \text{otherwise} \end{cases} \]

The derivative of the output with respect to the parameters is 0, as there are no parameters in the ReLU layer, making it impossible to learn.

No parameters, nothing to learn.
Multiple blocks

To update $w_2$

$$\frac{\partial o_2}{\partial w_2}$$

To update $w_1$

$$\frac{\partial o_2}{\partial w_1} = \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1}$$

Each block has its own:
- Derivative w.r.t. input
- Derivative w.r.t. parameters.

When you are back-propagating, be careful which one to use.
Multiple blocks

\[
f_1(x; w_1) \xrightarrow{o_1} f_2(o_1; w_2) \xrightarrow{o_2} f_3(o_2; w_3) \xrightarrow{o_3}
\]

\[
\frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1}
\]
Multiple blocks

\[ f_1(x; w_1) \rightarrow o_1 \rightarrow f_2(o_1; w_2) \rightarrow o_2 \rightarrow f_3(o_2; w_3) \rightarrow o_3 \]

\[ \frac{\partial o_3}{\partial w_1} = \frac{\partial o_3}{\partial o_2} \frac{\partial o_2}{\partial o_1} \frac{\partial o_1}{\partial w_1} \]

Last step: multiply with derivative w.r.t. parameters

Chain the “derivatives w.r.t. to input”
Exercise: work out all the details in a network where

- The input X is N-by-D (N examples)
- The label vector is N-by-K (in one-hot representation)
- There is a fully-connected layer and a softmax layer.
- Use cross-entropy as the loss.
$X \rightarrow XW \rightarrow \text{softmax} \rightarrow O_1 \rightarrow O_2 \rightarrow \text{Cross entropy} \rightarrow \text{Loss}$

$X: N \times D \& \ Y: N \times K \quad (K \text{ classes, one-hot rep.})$

$W: D \times K$

$O_1: N \times K \quad L: 1 \times 1$

$O_2: N \times K$

Learning takes place only in the first box.
\[
\frac{J_L}{\partial W} = \frac{J_L}{\partial o_2} \cdot \frac{\partial o_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial w}
\]

\(W\) is a matrix of weights. So, using "\(\partial\)" is an abuse of notation.
Derivative of Softmax

\[ q_i = \frac{e^{x_i}}{\sum_k e^{x_k}} = e^{x_i} \cdot \frac{1}{Z} \]

\[ \nabla q_i = e^{x_i} \cdot \frac{1}{Z^2} \cdot \sum_k e^{x_k} - 1 \cdot e^{x_i} = q_i - q_i^2 = q_i (1 - q_i) \]

\[ \nabla \nabla q_i = e^{x_i} \cdot \frac{1}{Z^2} \cdot e^{x_j} = -q_i q_j \]
Derivative of cross-entropy

\[ F = - \sum_k p_k \log q_k \quad \text{where} \quad q_k = \frac{e^{x_k}}{\sum e^{x_i}} \]

\[
\frac{\partial F}{\partial x_i} = - \sum_k p_k \frac{1}{q_k} \frac{\partial q_k}{\partial x_i} = \begin{cases} q_k (1-q_k) & \text{if } i = k \\ -q_i q_k & \text{otherwise} \end{cases}
\]
\[-\left(\frac{p_i}{q_i} \cdot q_i (1-q_i)\right) - \sum_{k+i} \frac{p_k}{q_k} (-q_i q_i)\]

\[-(p_i - p_i q_i) + \sum_{k+i} p_k q_i\]

\[-p_i + \sum_k p_k q_i\]

\[-p_i + q_i \sum_k p_k \cdot 1 = q_i - p_i\]
Convolutional layer

- Can be implemented using matrix multiplication (instead of convolution)
  - Pros: we can make use of highly efficient linear algebra packages (e.g. BLAS, OpenBLAS)
  - Cons: same filter is repeated many times in the matrix → a lot of memory required

[example on board]
Convolutional layer as matrix multiplication

Example: suppose the input is 200x200 color images, hence 200x200x3

Conv layer: 75 filters of size 10x10x3. Stride 2

Take every block of 10x10x3 on the input image and turn it into a 300x1 vector.

Collect all such blocks and stack them in a matrix.
Convolutional layer as matrix multiplication

\[(200-10)/2 = 80\text{ locations in total.}\]

So, 80x80 300-dim vectors. \(X = 300 \times 6400\)

Conv filters: 75 filters of size 10x10x3: vectorize each filter and stack them row-by-row \(W = 75 \times 300\)

\(W \times X \rightarrow 75 \times 6400\). Reshape this back to size (?)
Convolutional layer as matrix multiplication

\[(200-10)/2 = 80\] locations in total.
So, \(80\times80\) 300-dim vectors. \(\rightarrow X = 300\times6400\)

Conv filters: 75 filters of size \(10\times10\times3\): vectorize each filter and stack them row-by-row \(\rightarrow W = 75\times300\)

\(W\times X \rightarrow 75\times6400\). Reshape this back to size \(80\times80\times75\)
On the initialization of NNs
• When the number of examples is not large enough, unsupervised pre-training helps.
  – This was done in Mohamed et al. (2009) and Krizhevsky et al. (2012).
    • Add a new layer, initialize it using an unsupervised method such as Restricted Boltzman Machines or Autoencoders.
    • After adding all layers, put a supervised layer on top (e.g. softmax + cross-entropy), then continue training

• Since then, many papers on initialization. [e.g. Glorot & Bengio (2010); He et al. (2015)]
It is reasonable to assume that in the final trained network, half of the weights will be positive, and half of them will be negative.

- So, starting with all zeros might sound like a good idea.

It is reasonable to assume that in the final trained network, half of the weights will be positive, and half of them will be negative.

- So, starting with all zeros might sound like a good idea.
  - No, it’s not!! Why?

• How about initializing with small random numbers?
  - e.g. \( W = 0.01 \times \text{np.random.randn}(D,H) \) where \( \text{randn()} \) samples from the normal distribution \( N(0,1) \).
  - Good because it breaks the symmetry (i.e. the problem with starting with all zeros).
  - But it could lead to two different problems:
• How about initializing with small random numbers?
  - e.g. $W = 0.01 \times \text{np.random.randn}(D,H)$ where `randn()` samples from the normal distribution $N(0,1)$.
  - Good because it breaks the symmetry (i.e. the problem with starting with all zeros).
  - But it could lead to two different problems:
    • Gradients might vanish! (due to very small numbers)
    • The output from a randomly initialized neuron has a variance that grows with the number of inputs. (Could lead to very large responses!) Why?
Consider a perceptron with weights \( \mathbf{w} \) and input \( \mathbf{x} \).

We are interested in \( \text{Var}(\mathbf{w}^T \mathbf{x}) = ? \)

We are working with \( \mathbf{w} \) are indep. random variables, with small variances.

\( \mathbf{x} \) is a constant vector, so \( \mathbf{w}^T \mathbf{x} \) is a sum of normal vars, each scaled with a positive or a neg. number.

Remember: \( \text{Var}(a \mathbf{x}) = a^2 \sigma^2 \) where \( \text{Var}(\mathbf{x}) = \sigma^2 \)

Also \( \text{Var}(\mathbf{x} + \mathbf{y}) = \sigma^2 + \beta^2 \) where \( \text{Var}(\mathbf{y}) = \beta^2 \)

\[ \text{Proof: } \text{Var}(\mathbf{x} - \mathbf{y}) = ? \]

we know \( \text{Var}(\mathbf{x}) = \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])^2] \)

\[ \mathbb{E}[(\mathbf{x} - \mathbf{y})^2] - (\mathbb{E}[\mathbf{x} - \mathbf{y}])^2 = \mathbb{E}[(\mathbf{x}^2 - 2 \mathbf{x} \mathbf{y} + \mathbf{y}^2)] - (\mathbb{E}[\mathbf{x}] - \mathbb{E}[\mathbf{y}])^2 \]

\[ = \mathbb{E}[(\mathbf{x} - \mathbb{E}[\mathbf{x}])^2 + (\mathbf{y} - \mathbb{E}[\mathbf{y}])^2 - 2 \mathbb{E}[\mathbf{x}] \mathbb{E}[\mathbf{y}].] \]

\[ \text{indep } \mathbf{x}, \mathbf{y} \]

\[ = \text{Var}(\mathbf{x}) + \text{Var}(\mathbf{y}) \]

So with more dimensions \( \text{Var}(\mathbf{w}^T \mathbf{x}) \) increases.
• Solution: calibrate the variance:
  - Use \( w = \frac{\text{np.random.randn}(n)}{\sqrt{n}} \) where \( n \) is the number of inputs to the layer.
  - Xavier initialization [Gloroth and Bengio (2010)]
    • Sample weights from \( N(0, \text{Var}(W)) \) where

\[
\text{Var}(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}}
\]

Since the introduction of ReLU, initialization problem seems to have been solved:

- In He et al. (2015),

\[ w = \text{np.random.randn}(n) \times \sqrt{2.0/n} \]

is suggested.
- This is the current recommendation used in practice.

Also, with Batch Normalization (next slide) networks become less sensitive to initialization.

Batch normalization
Basically, normalizes the input (i.e. activations of the previous layer) at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1.
Batch Normalization

- Applying this technique usually amounts to inserting the BatchNorm layer immediately after fully connected or convolutional layers before non-linearities.
- It has become a very common practice to use BatchNorm.
- Networks that use BatchNorm are significantly more robust to bad initialization.
- BatchNorm also regularizes the model: “in a batch-normalized network we found that Dropout can be either removed or reduced in strength” [Ioffe and Szegedy (2015)]
Fine-tuning: another initialization method

• Typical scenario:
  – You want to train a network on your dataset $D$.
  – Grab an already trained network $N$. $N$ was trained on a different dataset than $D$ (but still in the same domain, e.g. natural images).
  – Use $N$ as your initial network and continue training (with very small learning rates).

• This is done a lot in computer vision.
  – e.g. Faster RCNN object detector starts its training from a network that was trained on ImageNET.
Stochastic gradient descent variants
Stochastic Gradient Descent (SGD)

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration $k$

Require: Learning rate $\epsilon_k$.
Require: Initial parameter $\theta$

while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.
    Compute gradient estimate: $\hat{\nabla} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
    Apply update: $\theta \leftarrow \theta - \epsilon \hat{\nabla}$
end while

[From Goodfellow et al. (2016)]
Setting the learning rate ($\alpha$)

- **Constant learning rate:**
  - When it is low enough, guaranteed to make zero or positive progress.

- **Decaying learning rate:**
  - The rate is decreased as a function of the iteration number.
  - **Decaying is desired** because the gradient estimate is noisy due to the random sampling of $m$ training examples. This noise does not vanish even when we arrive at a minimum.
  - By decreasing the rate, this source of noise is progressively eliminated.
Setting the learning rate ($\alpha$)

- Decaying learning rate:
  - Step decay
    - Reduce the rate by some factor every few epochs.
    - E.g. Halve the rate every 5 epochs → Numbers depend on the problem and dataset.
  - Exponential decay
    \[ \alpha = \alpha_0 e^{-kt} \]
    - $t$: iteration number
    - $\alpha_0, k$: hyperparameters
  - 1/t decay
    \[ \alpha = \frac{\alpha_0}{1 + kt} \]

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate ($\alpha$)

• Practical tips:
  - Step decay is slightly more preferable b/c its decrease schedule is more interpretable.
  - If you can afford, do slower decay and train for a longer time.

[From http://cs231n.github.io/neural-networks-3/]

22.10.2019
Setting the learning rate \((\alpha)\)

- There are also second order methods:

\[ w^{t+1} \leftarrow w^t - \alpha \nabla_w f(x; w) \]

where alpha is a function of the Hessian of \(f()\).

Newton method, BFGS, L-BFGS, Conjugate gradient

Not commonly used in deep learning.
Setting the learning rate ($\alpha$)

Practical tips

Adjust the hyper-parameters of your decaying schedule to get a “good” curve

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate ($\alpha$)

Practical tips

Adjust the hyper-parameters of your decaying schedule to get a “good” curve

[From http://cs231n.github.io/neural-networks-3/]

loss

very high learning rate

decaying schedule

low learning rate

high learning rate

good learning rate

epoch
Setting the learning rate \( (\alpha) \)

- A real example: training on CIFAR-10 dataset.
- Looks reasonable (maybe a little bit low alpha)
- Batch size might be a little too small (cost is very noisy).

[From http://cs231n.github.io/neural-networks-3/]
Setting the learning rate turned out to be really difficult.

So, researchers have develop adaptive learning rate methods
Adaptive learning rate methods

Momentum:

\[ v = m v - \alpha \nabla_w f(w) \]

Update: \[ w = w + v \]

Typical value for momentum is \( m=0.9 \)
Nesterov momentum

\[ w^{\text{ahead}} = w + mv \]

\[ v = mv - \alpha \nabla_{w^{\text{ahead}}} f(w^{\text{ahead}}) \]

Update: \( w = w + v \)

Works slightly better than plain momentum.

[From http://cs231n.github.io/neural-networks-3/]
Per parameter adaptive learning rate methods

• So far, learning rate was global (equally applied to all parameters in the model)

• Methods have been proposed to adapt learning rates per parameter.
  - Motivation: the cost function is highly sensitive in some directions and insensitive in others, in the parameter space
  - So, it might make sense to use different learning rates per parameter.
Per parameter adaptive learning rate methods

- E.g. “delta-bar-delta” algorithm [Jacobs (1988)]:

  “A heuristic method.
  If the partial w.r.t. to a given model parameter, remains the same sign, then the learning rate should increase.
  If it changes sign, then the learning rate should decrease. “

  [Goodfellow et al. (2016)]
Per parameter adaptive learning rate methods

- AdaGrad
- RMSprop
- Adam
Adagrad  [Duchi et al. (2011)]

```python
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

[From http://cs231n.github.io/neural-networks-3/]

Cache keeps track of per parameter sum of squared gradients.

Weights with high gradients $\rightarrow$ decrease learning rate
weights that receive small or infrequent updates $\rightarrow$ increase learning rates.

AdaGrad performs well for some but not all deep learning models.
**RMSprop**

[Slide 29, Lecture 6, Hinton's Coursera class]

```python
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

[From http://cs231n.github.io/neural-networks-3/]

Modifies Adagrad to use a moving average of squared gradients (instead of the complete sum over history).

Typically, `decay_rate = 0.9`

Performs well.
Adam [Kingma and Ba (2014)]

\[
m = \beta_1 m + (1 - \beta_1) dx \\
v = \beta_2 v + (1 - \beta_2) (dx**2) \\
x += -learning\_rate \times m / (np.sqrt(v) + \text{eps})
\]

[From http://cs231n.github.io/neural-networks-3/]

Uses a “smoothed” gradient \( m \).

Recommended values for hyper-parameters: \( \text{eps}=1e-8 \), \( \beta_1=0.9 \), \( \beta_2=0.999 \)

In practice, Adam is currently recommended as the default method to use.

[From http://cs231n.github.io/neural-networks-3/]
There are many more adaptive learning methods.

For a good overview, see

“An overview of gradient descent optimization algorithms” by Sebastian Ruder: link
This is an active area of research. Some recent methods:

- **Cyclical learning rates** [Smith, WACV 2017]

  "increasing the learning rate might have a short term negative effect and yet achieve a longer term beneficial effect."

  "increasing the learning rate can also allow for more rapid traversal of saddle point plateaus."

[Figure from https://www.jeremyjordan.me/nn-learning-rate/]
Stochastic Gradient Descent with Warm Restarts

[Loshchilov, Hutter ICLR 2017]

Escapes local minima by large LR at each restart.
Can be used to obtain an ensemble model.

[Figure from https://www.jeremyjordan.me/nn-learning-rate/]
“Don't Decay the Learning Rate, Increase the Batch Size” [Smith et al. ICLR 2018]

“ we ... obtain the same learning curve on both training and test sets by instead increasing the batch size during training. “

“ It reaches equivalent test accuracies after the same number of training epochs, but with fewer parameter updates, leading to greater parallelism and shorter training times.”
References

References


- Smith SL, Kindermans PJ, Ying C, Le QV. Don't decay the learning rate, increase the batch size. ICLR 2018.